frontiers in Computing

Preprints of the
International Conference
held from 9 to 11 December 1987, in Amsterdam,
The Netherlands

edited by

R.P. van de Riet

FRONTIERS IN COMPUTING

Preprints of the
International Conference
held from 9 to 11 December 1987, in Amsterdam,
The Netherlands

edited by

R.P. van de Riet

Vrije Universiteit

Amsterdam, The Netherlands

© Elsevier Science Publishers B.V., 1987

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owner.

The contents of this volume will be published as part of the journal FUTURE GENERATIONS COMPUTER SYSTEMS.

Compiled by Judith Clement
November 1987

PREFACE

It must have been late 1981 that I heard for the first time of the Fifth Generation Project in the form of a very ambitious plan which a director of an Industrial Research Laboratory showed me. He could not believe that the plan could be realised at all.

The plan was presented well in an international conference, the first on FGCS. It caused quite an explosion of political and research activities. major research institutes were getting impetus, like the Japanese ICOT, the American MCC, the European ECRC, and also the Dutch CWI, while national and international programmes were set up, like Alvey in the U.K., SPIN in the Netherlands, the German 'Informationstechnik', ESPRIT in Europe, just to mention a few important examples.

It was all triggered by a Japanese committee under the chairmanship of

Professor Tohru Moto-oka.

He had the vision to set up a ten-year plan with very ambitious goals:

- scientifically, in the form of a new computer concept, replacing the von Neumann architecture, exploiting parallelism in the form of very sophisticated applications with knowledge base architecture
- economically, by giving the Japanese industry new products to export, based on intelligence, amply available in the densely populated Japan
- socially, in providing the Japanese people access to knowledge in their homes, like water and electricity, and also providing new employment opportunities.

As a result of Moto-oka's plans conferences were organized. The first in which I directly involved was a one-day seminar at my University, in May 1983, the second was an international conference in December 1985 in Rotterdam. The present conference is thus the third one. These conferences are closely linked with the journal Future Generation Computer Systems (FGCS), of which Einar Fredriksson is the Publisher. The co-chairmen of this conference: ***Hideo Aiso*** (Keio University, Yokohama, Japan), ***Frank Kuo*** (SRI International, Menlo Park, USA) and myself are the consulting editors of FGCS. Because of his great expertise in the subject matter of our conference we have also asked ***Bob Hertzberger*** (University of Amsterdam) to join the group of co-chairmen.

The programme of this conference is dedicated for one half to invited speakers which can give overviews of results of coordinated research programmes and one half for submitted papers. The programme committee (with the help of 53 referees) selected 14 outstanding papers from 30 papers. These papers are mostly in the area of parallel architectures.

A strong emphasis is also given to the issue of comparing research programmes of the various institutes and cooperations. In this context a strong panel has been organized by Mr. Friedrich Winkelhage (GMD, FRG).

The issues of the conference were originally set up as follows:

- Non-Conventional Architectures
- Architectures for Neuro- and Supercomputing
  - Bio- and Molecular Components
  - Optical Components,
- New Architectures for large systems: Networks
- Computing needs of Society and the impact on software
- Comparison of IT programmes

In order to honour the great work of prof. Moto-oka who, to our great regret, died in the age of 56, on November 11, 1985, we decided to organize in this conference two **Moto-oka Memorial sessions.**

As has been said above, his work was of crucial importance for developing new computer architectures, in particular the Fifth Generation Computer. He had the foresight of a new computer system in which new hardware techniques (VLSI) and sophisticated software developments (Logic Programming) are genuinely combined into intelligent systems.

In these special sessions the following speakers will highlight the role of Prof. Moto-oka in the research programmes of their respective countries and institutes:

1. **Prof. Hideo Aiso, Keio University,** Yokohama, who is the successor of Prof. Moto-oka as Chairman of the Fifth Generation Project as developed by the **ICOT** institute in Tokyo.
2. **Dr. John T. Pinkston**, Chief Scientist and Vice-President of the **Micro-electronics and Computer Technology Corporation (MCC)** in Austin Texas.
3. **Dr. Jean-Marie Cadiou**, director of the European **ESPRIT** programme, who will speak on the second phase of ESPRIT.
4. **Dr. Sunichi Uchida**, chief researcher of the **ICOT** Institute, who will speak on the major results ICOT's research has established in the area of parallelism.

We are proud and happy with this conference bringing together researchers from so many countries all cooperating and interested in the challenging problems around software and hardware of computers of the next generation. I would like to close with one quotation of Prof. Moto-oka, where he speaks about international cooperation; during the First International Conference on FGCS in Tokyo, October 1981 he said:

"We have to be internationally competitive and at the same time cooperative in the following subjects:

1. Construction and maintenance of various data bases;
2. Smoothing international exchanges through the development of translation-assisting and interpreting systems;
3. Improvement of productivity with the aid of intelligent robots; and
4. Accelerating research and development by using intelligent CAD systems.

Although we have mainly followed the head of other countries in computer technology up to now, it's time for us to break with this outmoded tradition, and center our efforts on the development of new computer technology based on our own conceptions, so that we can provide the world with new technology with a view to promoting international cooperation".

I think that the papers in these preprints prove that the director, I referred to, was not right.

Prof. Dr. R.P. van de Riet
General Chairman
(Vrije Universiteit, Amsterdam, The Netherlands)

COMMITTEES

Conference Chairman
R.P. van de Riet
(Free University, Amsterdam, The Netherlands)

General Co-Chairmen
H. Aiso
(Keio University, Yokohama, Japan)
L.O. Hertzberger
(University of Amsterdam, Amsterdam,The Netherlands)
F. Kuo
(SRI International, Menlo Park, USA)

Programme Committee
H. Aiso
(Keio University, Yokohama, Japan)
M. Amamiya
(NTT Research Laboratories, Tokyo, Japan)
P.M.G. Apers
(Twente University, Enschede, The Netherlands)
F. L. Carter
(Naval Research Laboratory, Washington, USA)
H.J. Caulfield
(University of Alabama, Huntsville, USA)
J.Fox
(CALTEC, Pasadena, USA)
G. Goos
(GMD, Karlsruhe, FRG)
J. Gurd
(Manchester University, Manchester, UK)
L.O. Hertzberger
(University of Amsterdam, Amsterdam, The Netherlands)
H. Hünke
(ESPRIT Task Force, Brussels, Belgium)
H. Kashiwagi
(ETL, Ibaraki, Japan)
R. M. Keller
(Quintus Computer Systems, Mountain View, USA)
H.T. Kung
(Carnegie Mellon University, Pittsburgh, USA)
F. Kuo
(SRI International, Menlo Park, USA)
G. Matsumoto
(ETL, Ibaraki, Japan)
Y. Muraoka
(Waseda University, Tokyo, Japan)
L. Nijman
(Philips Research Laboratories,
Eindhoven, The Netherlands)
R. P. van de Riet (chairman)
(Free University, Amsterdam, The Netherlands)
S. Shibayama
(Toshiba R&D Center, Kawasaki City, Japan)
H. Tanaka
(University of Tokyo, Tokyo, Japan)

P. Treleaven
(University College London, London, UK)
S. Tsur
(MCC, Austin, USA)
S. Uchida
(ICOT, Tokyo, Japan)
B.P.Th. Veltman
(Delft University of Technology, Delft, The Netherlands)
F. Winkelhage
(GMD, St. Augustin, FRG)

Honorary Board
P.C. Baayen
(CWI, Amsterdam, The Netherlands)
A. Bensoussan
(INRIA, Le Chesnay, France)
J.-M. Cadiou
(CEC/ESPRIT, Brussels, Belgium)
M.C.G. Hartmann
(Ministry of Economic Affairs, The Hague, The Netherlands)
J.A. Ionson
(OSD/SDIO, Washington, USA)
H. Kashiwagi
(ETL, Ibaraki, Japan)
B. Oakley
(Alvey Directorate, London, UK)
J.T. Pinkston
(MCC, Austin, USA)
F. Winkelhage
(GMD, St. Augustin, FRG)

Organizing Committee
E.H. Fredriksson (chairman)
(Elsevier Science Publishers, Amsterdam, The Netherlands)
H. Bosma
(Philips Research, Eindhoven, The Netherlands)
J.C.P. Bus
(CWI, Amsterdam, The Netherlands)
E.J. van de Kraats
(Shell International Research, Rijswijk, The Netherlands)
A. Sevenster
(Elseviers Science Publishers, Amsterdam, The Netherlands)
B.P.Th. Veltman
(Delft University of Technology, Delft, The Netherlands)
W.K. Vietsch
(Ministry of Education and Science, The Hague, The Netherlands)

The REFEREES

The following persons have been very helpful in the refereeing process:

M.S. Aistrope

J.K. Annot

L. Augusteyn

Mr. Bal

B. Nanieqbal

H. Barringer

M. Boaron

A.P.W. Böhm

S. Brookes

J. Burghardt

E. Castrinakis

J.F. Foley

T. Furuya

J. Grosch

K. Hirata

P. Highnam

H. ter Horst

Y. Ikemoto

S. Jähnichen

M.L. Kersten

M. Kohata

A. Kokubu

Mr. Landwehr

J. van Loon

W.G.P. Mooij

J.P. Morrison

M. Nagatami

T. Ogura

Mr. Ohlu

J. Omnes

R. Onai

E.H.J. Persoon

Mr. Rudolph

S. Sakai

A.G. Starreveld

M.A. Standring

L. Steels

Y. Sugito

N. Takahashi

Mr. Tebra

K. Toda

Y. Uchibori

A.H. Veen

F. Victor

Mr. de Vink

W.G. Vree

W.E. van Waning

I. Watson

P. Watson

Mr. Weigand

Mr. Wieringa

T. Yuba

W. Zimmermann

PROGRAMME AND TABLE OF CONTENTS

Preface
by Prof.dr. R.P. van de Riet, Conference Chairman v

Committees xii

The Referees ix

Moto-oka memorial session I
chairman: *R.P.van de Riet*
The Fifth Generation Project in Japan *H. Aiso* ..... *
From Visionary Ideas to Products *J. T.Pinkston* 1

Moto-oka memorial session II
chairman: *L.O. Hertzberger*
The Second Phase of ESPRIT *J. M. Cadiou* ..... *
Recent Progress of the Parallel Inference Machines at ICOT *S. Uchida* ..... *

Panel on Comparison of IT Programmes
chairman: *F. Winkelhage*
H.Aiso ..... *
P.C. Baayen : Policy, Program and Perspectives of CWI, Amsterdam 17
J.M. Cadiou ..... *
D. Cohen ..... *
H. Gallaire : ECRC : A Joint Industrial Research Centre 29
B. Oakley ..... *
J. T. Pinkston 1

Data Flow Machines
chairman: *E.A.M. Odijk*
Data Flow Computing and Parallel Reduction Machine *M. Amamiya* 37
Performance Issues in Dataflow Machines *J. Gurd , W. Bohm, Y.M.Teo* 53

Computing Needs for Society and the Impact on Software
chairman: *G. Goos*
Software Developments and SDI *D. Cohen* ..... *
Current Status of JUNET *J. Murai , A. Kato* 69

Large Computer Networks
chairman: *F. Kuo*
Self-organizing Networks *M. Frankel , N. Schacham, J.E. Mathis* 83
Routing Management in Very Large-Scale Networks *J. Garcia-Luna* 109

Panel on Future Networks for Research
chairman: *F. Kuo*
M. Frankel
J. Garcia-Luna
L. Landweber
D. Jennings
P. Linington
R. Kahn
J. Murai
P. Tindemans
W. Zander

VLSI Implementations
chairman: *B.P.Th.Veltman*
3-D VLSI Technology in Japan and an Example: a Syndrome Decoder for
Double Error Correction *T.Kurokawa, H. Aiso* 125
Optical Link in the Delft Parallel Processor : an Example of MOMI-Connection in
MIMD-Supercomputers *L.Dekker, E.E.E. Frietman, W. Smit, J.C. Zuidervaart* 141

Parallel Architectures for High-level Operations on Sets and Databases
chairman: *P.Apers*
A Parallel Machine for Multiset Transformation and its Programming Style
J.-P.Banâtre, A. Coutant, D. Le Métayer 157
Cellular Array Architecture for Relational Database Implementation *M. R. Muraszkiewicz* 175

Parallel Architectures for Artificial Intelligence I
chairman: *R.M.Keller*
DLM- A Powerful AI Computer for Embedded Expert Systems *A.R.G.Pudner* 187
Wave-1: A New Ideology of Parallel and Distributed Processing on Graphs and Networks
P.A. Sapaty 203

Parallel Architectures for Artificial Intelligence II
chairman: *S. Uchida*
Brave - A Parallel Logic Language for Artificial Intelligence
T.J.Reynolds, A.J. Beaumont, A.S.K. Cheng, S.A. Delgado-Rannauro, L.A. Spacek 221
Study of Mechanisms that Support the Implementation of Non-logical Components
of Prolog in WAM-based Systems *Z.Chenxi, T. Yungui* 237

Systolic Arrays
chairman: *J.Fox*
Systolic Arrays; the WARP machine *H.T. Kung* ..... *
Implementation and Performance of a Complex Vision System on a Systolic Array Machine
J.N. Webb, E. Clune, J.D. Crisman, G.J. Klinker 247

Optical Components
chairman: *H. Kashiwagi*
Optical Components for Digital Optical Circuits *S.D. Smith, B.S. Wherrett,*
F.A.P. Tooley, A.C. Walker 263
Optical Computing in Japan *T. Yatagai* ..... *

Parallel Reduction Machines
chairman: *P. Treleaven*
A Reduction Architecture for the Optimal Scheduling of Binary Trees
K.Ravikanth, P.S. Sastry, K.R. Ramakrishnan, Y.V. Ventakesh 273
The Dutch Parallel Reduction Machine *H.P.Barendregt, M.C.J.D. van Eekelen,*
M.J. Plasmeijer, P.H. Hartel, L.O. Hertzberger, W.G. Vree 285

VLSI and Advanced Systems
chairman: *S. Tsur*
A Recursively Scalable Network VLSI Implementation *G. Della Vecchia, C. Sanges* 299
A VLSI Implementation of an Architecture for Applicative Programming *T.Bridges,*
S.W. Kitchel, J.T. O'Donnell 315

Beyond the Frontiers in Computing
chairman: *H. Aiso*
Logic Programs Directly Processed in a Network of Content Addressable Memories
A.de Callataÿ 331
Neurocomputing *G. Matsumoto* ..... *

List of Authors 347

* not received

FROM VISIONARY IDEAS TO PRODUCTS

John T. Pinkston

Microelectronics and Computer
Technology Corporation (MCC)
Austin, Texas, USA

1. INTRODUCTION

It is a pleasure and an honor to be here with you today in this session of the Frontiers in Computing Conference, honoring Prof. Tohru Moto-Oka. It is very fitting that there should be a speaker from MCC at such a session, because it is at research institutes such as MCC that the great influence of Moto-Oka and his colleagues is apparent. Indeed, were it not for his vision, and the persuasive articulation of that vision, MCC would almost certainly never have been created, and the same is likely true also for many other of the research activities in our field today.

In this paper, I will first review the history of how the Microelectronics and Computer Technology Corporation (MCC) came to be created. This was done by a set of United States computer and microelectronics companies, as a cooperative research consortium, stimulated by the bold Japanese goals of the Fifth Generation and Supercomputer programs. Following this, I will describe where the MCC research programs stand today, and the results that are now beginning to appear. Finally, I will discuss some of the directions in which I see our fields of research moving.

2. THE RESEARCH CONSORTIUM CONCEPT

As technical people in the computer field, most of this audience will recognize the benefits of research, whose results are now incorporated into the hardware and software systems that are in use every day. You know that such research is necessary. The systems we enjoy today would not exist without the fundamental research that has gone on during the past two or three decades.

In carrying out this work of developing technology for improved or future products, executives are faced with many decisions. They must decide what areas to work in, and what efforts to pursue. They must decide on their time horizons, or the mix of long and short term work. They must decide what resources to devote, or even whether to invest at all, particularly in the longer range efforts.

There are many obstacles that deter investment in long range research. It is expensive, particularly to do right by devoting

sufficient resources to mount a critical mass on the problem. It has risk, and does not pay off quickly, and over the past decade, the trend has been for managers to be pressed to show profits in the near term and to avoid risk. And yet, while cost and risk are certainly factors, I believe that by themselves they simply don't tell the whole story. There are plenty of companies well able to shoulder substantial costs of research work when they choose to do so. There are plenty of astute technical managers who can evaluate risks, and who don't shrink back from taking them. And I have met too many far-sighted executives to explain the effect just by lack of vision.

There are three factors that I do believe are significant in deterring a rational manager from investing in advanced research in the world we operate in today. FIRST, it is becoming increasingly difficult for someone to keep the benefits of his research for himself -- and relatively easy for someone to catch up after letting others do the research work. For many companies this seems to be a deliberate policy -- letting the government or IBM break the ground and take the risks.

Certainly there has been much good computer science work done at Xerox PARC that has found its way into many personal computer products, probably the most notable being the Apple Macintosh line. Sometimes, as in the case of the Macintosh, the benefitting company takes the technology into a product in a new area, while in other cases there is a game of catch-up to bring comparable products to market after a pioneer such as IBM has shown the way.

In either case, the extent to which competitors can reap benefits without paying for them decreases the payoff in the calculation of the executive approving research expenditures. Further, in our mobile and open society, the longer the duration of the research program, the more the difficulty of keeping the results increases, and the greater the reluctance to fund such work. Thus there is an unstable situation in which everyone knows that continuing research is needed for technology to advance, and there is no real doubt that the results of such work will pay off, but each is waiting for the others to do it. Nobody wants to pay for the research work, only to have someone else take it and catch up quickly, or worse yet, win the race to manufacturing with copycat products that reap a share of the benefits.

A SECOND significant deterrent to a management decision to invest in research is that there is often a critical mass threshold. Researchers tend to work better in groups, with colleagues to talk to, exchanging and testing their ideas. There are economies of scale in equipment and facilities. Developments in one area may require concurrent advances in other related areas to really be useful. For example, a new high speed, high density semiconductor device technology would require new and improved packaging and interconnection techniques to keep from giving up at the system level the performance improvements that the devices promise.

A research manager of a non-giant company is thus faced with a dilemma -- he can stay within his company's areas of expertise for example the semiconductor device technology, and hope that the

other aspects such as packaging will be addressed by somebody else, and further, will be done soon enough and in the right form to be mated with the technology he is developing. Or he can try to develop all of the necessary parts himself, with the added cost and risk that comes from moving outside areas of traditional expertise.

A THIRD deterring factor is that longer range research tends to pay off better for big companies than for the small or medium sized ones. The problem is that the cost of obtaining the research results tends to be a constant, while the larger company has larger markets to generate the volume to recover costs. The giant company also has a better chance of finding opportunities in its broader product lines to exploit the unanticipated discoveries that often come from research, and a larger manufacturing infrastructure with which to move them to market.

Here is where cooperation becomes attractive. What one gives up is the possibility of being first with something unique, with whatever accompanying advantages would come with this. But even this is just a possibility, not a certainty, since many others are also working hard doing their comparable research. What one gains is the economy of sharing the cost. When one looks at the costs of doing a complete program right, the prospect of getting the results of an adequately-sized and funded effort for a fraction of the cost becomes quite attractive.

3. HISTORY OF MCC

The story of MCC really begins with a meeting that was called to discuss exactly these issues, held in Orlando, Florida, in February of 1982. It was hosted by William Norris, then chairman and CEO of Control Data Corporation, and was attended by senior executives from 16 computer and microelectronics companies.

These men were well aware of the problems associated with getting benefits from long term research and exploratory development, and with the somewhat deadlocked state of affairs currently existing. But now there was a new element to be considered, with the bold announcements by Moto-Oka and his colleagues of the very far-reaching and challenging goals of the Japanese Fifth Generation and Supercomputer programs.

These plans staked out new potential areas of competition, pushing into markets where U.S. companies previously had been virtually unchallenged in the United States, by announcing an intent to leap over and beyond current technology.

The goals were very ambitious; indeed many thought them unreachable and unrealistic. But they were to be funded at very impressive levels of $500 M for the Fifth Generation over 10 years, and $200 M for the Supercomputer program over 5 years. And they were taken very seriously, because if the goals articulated by Moto-Oka could be achieved, it would change the face of the information processing industry.

To understand fully the feelings that these announcements caused, one needs to go back in history to the 1970's, when Japan had mounted a similar program to those announced in 1982. This was the VLSI program, in which MITI supported and orchestrated an effort to take a leap forward in semiconductor manufacturing technologies. In 1982, it was becoming apparent that this program had succeeded greatly, and the trends in market penetration of Japanese integrated circuits, particularly dynamic RAMs, were becoming clear and alarming to the U.S. executives. By late 1981, Dataquest had estimated that the Japanese had captured over 70% of the worldwide market share for 64K DRAMs. The quality of the parts, as well as the price and functionality, had become a big issue, and the Japanese were winning points there also. It was clear that they had stolen a march on their U. S. competitors, and now enjoyed a powerful initiative that was likely to last a long time.

Then, on the heels of this success in the integrated circuit manufacturing area, came the announcements of their intentions to repeat that process in the computer sector. It is no wonder that after seeing the results of the VLSI program, the new initiatives were taken very seriously indeed by the U.S. industry. There was a mood of concern, even fear. It now appeared that as much as these companies might find it hard to afford to do the research needed to stay up with the international competition, they might find it even harder to afford NOT to do it if they were to remain competitive in the 1990's.

It was in this mood of concern that Norris saw his chance to accomplish a dream he had carried and pushed for years -- the concept of cooperation among companies in carrying out research that all will need in a joint, cost shared way.

Thus these men meeting in Orlando came to consider a solution to the challenges facing them might lie in cooperation, by performing the fundamental, basic research work in their areas of need as a joint effort. This would be research that each would otherwise have to do himself, or else wait for somebody else to do, with the first choice being expensive and risky, and the second implying giving up technological leadership and any ability to shape the directions in which the technology would move. A cooperative effort would achieve economies of sharing the cost, multiplying one's research dollar by the number of cooperating parties, avoiding wasteful and unnecessary duplication of effort, achieving the critical mass to obtain economies of scale, and sharing the risk.

There was sufficient agreement that this was a promising approach for these companies to form task forces made up of some of their technical experts to address the details of what research such a cooperative would undertake, and how it would go about it. So began the chain of events that culminated in the formation of MCC.

These task forces worked through the summer of 1982, visiting research laboratories around the country and talking to numerous other experts in the microelectronics and computer fields. By December of 1982, they had completed outlines of 4 research programs, in Computer Aided Design tools for VLSI circuits,

Software Technology, Microelectronic Packaging and Interconnect, and Advanced Computer Architectures. While all details were obviously not worked out, there was enough there to allow companies to make their decision as to whether or not to join this new consortium. Indeed history has shown that these task forces did their jobs very well, because as the programs have evolved, they have held, for the most part, to the directions that were set out at that time.

There were, of course, concerns also about the U.S. antitrust laws, in addition to the technical and business issues to be weighed in deciding to participate. But in December, 1982, the Justice Department cleared the way by issuing a statement that it would not challenge the creation of such a cooperative venture, although it would continue to watch the activities as they evolved. With this clearance, ten companies agreed to join at that time, and MCC was created. Three more companies came in almost immediately thereafter, and since then, others have joined, and some have departed, so that now the number stands at 20. Each member, which must be a U.S. company, buys one share of stock and agrees to participate in sharing the costs and results of at least one of the research programs.

The member companies as of this writing are Advanced Micro Devices, Allied-Signal Corp., Bell Communications Research, Boeing, Control Data, Digital Equipment, Eastman Kodak, General Electric, Harris, Hewlett-Packard, Honeywell, Lockheed, Martin Marietta Aerospace, 3M, Motorola, National Semiconductor, NCR, Rockwell, Unisys, and Westinghouse.

The next steps were to hire a chief executive and select a location for the new venture. The first of these was accomplished in January, 1983, with the recruiting of Bobby R. Inman. The site selection process was highly publicized, with 57 potential sites in 27 states making presentations to the site selection committee. This is probably some kind of record for the amount of fuss over the location of a company with one employee that had yet to make its first dollar of profit. Austin, Texas was selected in May on the basis of the proximity to a major research university, general quality of life, and a package of incentives offered by the state and local governments, business, and The University of Texas.

Operations began in leased quarters in September, when a handful of pioneers moved in and started the process of assembling staff and facilities, and making the detailed plans.

4. MCC TODAY

The organization has grown steadily to about 450 today as this is written, with 360 in the research programs, and of these, 310 being actual research scientists, engineers, and technicians.

There are four research programs: Advanced Computer Architectures, Software Technology, VLSI CAD tools, and Packaging/Interconnect. The Advanced Computer Architectures is the largest, and is divided

into three laboratories: Human-Machine Interfaces, Systems, and Artificial Intelligence.

MCC's research programs share a common goal to provide the technology necessary to make computers, applications, and processes faster, more reliable, and capable of performing more complex tasks at a higher level of quality and at much lower cost.

4.1. The Advanced Computer Architecture Program

The overall thrust of the ACA program is to develop technologies upon which the information processing systems of the 1990's can be based, enabling extraordinary increases in power (both speed and functionality) and in ease of use.

4.1.1. Human Interface Laboratory

The Human Interface Laboratory is developing technologies to enable people to accomplish tasks more effectively and comfortably with computers. In this project, the influence of Moto-Oka's writings is perhaps most apparent of all MCC programs. The theme of extreme ease of use permeates the writings of the Japanese Fifth Generation visionaries.

The full potential of future computer systems can be realized only if they support more flexible forms of interaction and intelligently adapt to the needs of their users. They need to be forgiving and accommodating -- user helpful rather than just user friendly. A person should be able to approach an unfamiliar next generation system and use it as easily as we today get into a rental car and drive it away.

The problem with current systems is that they are rigid, difficult to learn, and severely constrained in the types of interaction supported. They require that users spend too much time concerned with the task of interacting with the machine rather than with the real tasks they are employing the machine to accomplish.

The goal of the MCC Human Interface Lab is to enable the development of interfaces where the quality of interaction approaches the richness of human interaction. The work is divided into projects which are exploring user needs, intelligent assistants, multilingual natural language understanding, and management of the knowledge required for intelligent interfaces.

Various experimental application interfaces have been built and evaluated and initial tools for some aspects of interface implementation have been developed. Work has begun on an architecture to combine these tools into a powerful, integrated system for building adaptive, intelligent interfaces.

In media related work, a graphic interface construction kit has been developed and integrated with MCC's object oriented database system. A voice/graphics interface for VLSI circuit layout has been developed, yielding significant engineering time savings. Multimedia information management procedures have been constructed to support multimode interfaces.

The User Assistance project has developed a general architecture for interface advisory systems and explored its use with an experimental data analysis system called VSTAT. The Natural Language project has produced an initial version of its English understanding system and a set of toolkits to assist in the construction of the linguistic components of interface systems. In addition, the project has developed morphological rules sets covering several languages and successfully completed a proof-of-concept exercise. A Linguist's Lab, composed of development tools and systems to aid linguists' research of multilingual understanding systems, has been assembled.

4.1.2. Artificial Intelligence Laboratory

The term "artificial intelligence" has been so overworked in the past years that it has lost virtually all precise meaning. It refers to a collection of algorithms, programming techniques, and information or knowledge representation techniques, across a wide range of problem areas. These include expert systems, logic programming, pattern recognition, image recognition/understanding, speech recognition/understanding, natural language processing, machine learning, adaptive systems, theorem proving, neural networks, robotics, and others.

MCC does not attempt to work in all of these areas of AI. The Artificial Intelligence/Knowledge-Based Systems lab (AI/KBS) in MCC is addressing the identification and encoding of knowledge into a base to support human-like intelligent behavior, the development of systems for constructing large scale expert systems, and the development of powerful reasoning tools.

The overall objective of this Lab is to significantly expand the range of intelligent behavior that can be achieved by computer systems. As recognized by the Fifth Generation writers and others, results in this area will greatly increase the functionality as well as support interfaces that will open up the accessibility to such resources for a much broader segment of society than uses them now.

The projects within the AI Lab are: 1) Large Structured Knowledge Base, which is developing a very large "common sense" knowledge base that will be the basis on which future intelligent systems will be built; and 2) Expert System Technology, which researching reasoning and representation techniques, and is building a development environment for expert systems incorporating techniques for reasoning about facts that change over time.

The Large Structured Knowledge Base project is building its "common sense" knowledge base system (called CYC) by encoding a wide range of knowledge, such as that found in an encyclopedia. The common sense knowledge that is not explicitly stated, yet is necessary to fully understand the encyclopedia entries, is simultaneously being identified and added to the system. Future AI systems based on this knowledge base will then be able to fall back on general knowledge when encountering a problem not anticipated by the system designer.

This project has developed the basic structure, and is in the process of building the CYC knowledge base. This task has included developing representation formats, identifying and encoding explicit and implicit knowledge, developing tools for creation of large knowledge bases, and organizing the encoded knowledge in the CYC knowledge base.

The Expert System Technology project has developed a basic expert system development environment called PROTEUS. This system integrates both commonly used and advanced reasoning techniques, such as backward and forward chaining, truth maintenance concepts and explanation-based generalization. This integration makes the use of these techniques more feasible than a loose "toolkit" approach, and the multiple representation techniques allow flexibility to encode many types of knowledge.

This project has developed and released the PROTEUS expert system development environment to shareholders. This system is being used as the basis for several commercial quality expert systems under development by shareholders, and forms the basis of the first product from a shareholder company to reach the marketplace. It includes features such as a unique method for controlling dependency-directed backtracking and a unification of frames and truth maintenance techniques. For the version of PROTEUS currently under development, we are working on adding temporal reasoning, metalevel control, and rule compilation.

We have chosen to focus critical-mass efforts on these information representation and expert systems technologies as the most appropriate to the needs of our shareholders, and to any advanced system architecture we might produce. However, one should note also that there is work in other labs and programs that would be termed AI: the Natural Language efforts in the Human Interface Lab, Neural Networks and Logic Programming in the System Lab, and Analogical Reasoning work in both the CAD and Software programs.

4.1.3. Systems Technology Laboratory

Next generation systems, including knowledge-based systems, computer aided design (CAD) systems, and speech, vision and text-based systems, will have to reason with very large knowledge bases at very high speeds. The mission of the Systems Technology Laboratory is to anticipate the computational needs of such emerging system applications, and to create the technology for new programming and execution environments responsive to these needs. New data models, languages, and processing strategies are required, including new approaches to both parallel and sequential architectural models. The program focuses on both software and hardware technologies to make these advances.

The assumption here is that the data that users will want to work with will be much more complex and semantically rich in structure than today's tabular forms, and that it will often be of a size that is too large to fit in a computer's main memory.

The goal is to develop system architectures and database technologies that promise performance that is 10 to 100 times that

predicted for 1990 with normal evolution. We are targeting applications of knowledge based systems, design systems, and multimedia systems while continuing to support traditional applications in the MIS domain. A second objective is to develop, by the end of 1988, an Object-Oriented Database System separate from our main effort but with the potential for later integration.

The Systems Technology Lab is pursuing three system concepts simultaneously: 1) the Advanced Database System (ADBS); 2) the Object-Oriented Database System (ODBS); and 3) characterization of parallelism in knowledge based systems. The ADBS is based on logic as a common language for representing data, knowledge, and associated programs. and involves new approaches to languages, compiler theory and parallel machine architecture. The ODBS has been designed as an expansion to object-oriented programming tools, providing a rich object-modeling capability and facilities for managing persistent, sharable objects.

The work in high performance systems architectures starts with the characterization of the computational needs of knowledge based systems and leads to investigation of appropriate languages, algorithms, and architectural concepts, both sequential and parallel. This effort is expected to be an important part of the System Laboratory's work, although symbolic processing algorithms have not progressed to need very large compute power as quickly as anticipated a few years ago, and we are in the process of identifying applications that will drive the systems architectures work to its conclusions.

An architectural design of ADBS has been completed, including a formal specification of its high level Logic Data Language (LDL), an optimizing compiler for LDL, and a scalable, parallel database machine for executing LDL object code. The first experimental compiler is now running, and the database machine experiment (based on a 40-node parallel processor) is now beginning to produce data. A performance model for the database machine has been implemented to assist in refinement of the architecture.

The first, single workstation, version of ODBS was developed, demonstrated, and transferred to shareholders in April 1987. ODBS has been interfaced to the PROTEUS expert system development environment created by MCC's AI Lab, and also to POGO, an experimental graphical object representation system being developed in the Human Interface lab. The Proteus/ODBS combination is the first truly tight coupling of an object data base system to an AI application that we are aware of. The distributed version of ODBS is scheduled for transfer in 1988.

4.2. Software Technology Program

The mission of MCC's Software Technology Program is to develop tools and methods that will significantly improve the productivity of the software development process and the quality of the software product. The research program will accomplish this by focusing on the early software development stages of requirements definition, specification decisions, and design prior to coding, specifically addressing large complex systems requiring a team effort of

cooperating designers. It is in this stage of the software design process that design alternatives are investigated and software system specifications discussed.

The technology for designing and developing large, complex software systems is seriously lagging. The problem is compounded by the fact that most of the large future systems will be distributed, which adds to the complexity of software design. These software systems can be developed on a reasonable time scale only through the use of teams of designers working simultaneously. However, very little work has been done on how such teams can work together and on what kind of computer support they will require.

The Software Technology program has been divided into four groups: 1) Design Information, finding ways to identify, represent, and organize in machine-processable form the kinds of information used in software design; 2) Design Process, analyzing the process of design, and developing computer aids to support it; 3) Design Interface, developing the technology to help designers, and especially those working in teams, interact effectively with one another and with the design technology; 4) Design Environments, uniting all the research into an integrated system.

The current emphasis of the Software Program is on the integration of research results. The main vehicle for carrying out this integration is a design environment called DELI, whose initial focus is on a network window manager which will allow tasks to be performed using network resources as if they were local to one machine instead of many. A new HyperText language called PlaneText/Fig has been developed to facilitate design activity. DELI is continuing this work toward the next generation of HyperText systems as well as developing new ways to store and retrieve software objects.

The program has developed an Object Server, a database capable of storing and retrieving organized data and processes for reuse in design, which has been used as a test bed for some experimental studies as well as for applications. A prototype reuse system, ROSE, has been developed, which allows system characteristics to be defined such that pattern matching and inferencing can be used to find analogous systems that have been designed previously.
In the Design Process group, the focus is on formal methods of system definition which will allow high-level descriptions of functions which can then be machine-implemented. Finally, in the Design Interface Group, the activities are based around team coordination. Emphasis in Design Interface is on how to use software to augment meetings and aid team interaction across the network. Also, advanced methods of visually representing software processes are being investigated.

I will go more quickly over the remaining two programs, although they are just as exciting and productive as the Advanced Architectures and the Software programs, because they are probably of less interest to this audience.

4.3 VLSI/CAD Program

As integrated circuit manufacturing technology advances permit the fabrication of circuits with millions or even tens of millions of devices, the design of the complex circuits that are now possible to fabricate is becoming increas- ingly difficult and time-consuming. In fact, it is probably fair to say that today we can fabricate chips that are more complex than we can design. A CAD system that employs novel approaches to increasing designer productivity is needed.

The VLSI/CAD Program will create a computer-assistance system which will support the design of very large testable integrated circuits. The near term goal of the research program is to develop an advanced software platform for CAD tool development, essentially a distributed, heterogeneous operating system. This software environment allows all designers using the CAD system to easily and uniformly access design information and tools. Longer term objectives include developing methods and tools for circuit design that advance the state-of-the-art in CAD to support the design of systems of ten million devices. This system must be a truly integrated design environment into which new application tools can quickly be added.

The CAD system will include a data management capability that supports maximum reuse of design information; employs efficient data structures for all required design representations; and incorporates a knowledge-based paradigm to help identify previous design information that is relevant to the current design task.

The VLSI/CAD Program has already delivered numerous software packages to the CAD Program participants, including a beta test version of a large integrated CAD system. This system contains an initial version of the Core Environment, with a high-performance graphics subsystem and editor common to all tools. A knowledge-based data structure (KBDS) scheme has been devised and tool integration into the platform is underway.

An interesting point to note is how similar are the issues in CAD to those addressed by the Software program. Both deal with the management of complexity of large design processes. The fact that one aims at executable code as an output (ones and zeros in a computer memory), and the other maps its output on to patterns on a glass plate, turns out to be more of a detail than a fundamental difference.

4.4 Packaging/Interconnect Program

The Packaging and Interconnect Program at MCC is a multi-disciplinary research effort which will enable electronic packaging to keep pace with the rapid advances in chip integration and speed, with substantial cost and performance improvements over current packaging and interconnection technologies. Research focuses on substrate materials, chip attachment techniques, cooling methods, and manufacturing. The challenge is to imbed higher speed and higher pincount integrated circuits into systems without giving up their performance at the system level.

Research carried out in the this program will provide the technology to support integrated circuits of the 1990's, which will include chip areas up to four square centimeters, chip power of up to forty watts, and signal rise times less than one hundred picoseconds. In order to minimize the risks for shareholders, MCC will assure the availability of all elements essential to its interconnect strategy, including connectors, materials, test and cooling technology, and process reliability.

Several advances in the state of the art will be transferred to shareholders by the end of 1987. In the area of Tape-Automated Bonding technology, we have demonstrated bonding of chips with up to 328 leads with 4 mil centers. Laser bonding technology has been developed for large variety of metallurgies at a rate of five bonds per second. Interconnect technology advances include a multilayer copper polyimide interconnect system and the design of a high density, flexible, controlled impedance connector. In the systems area, we have developed low cost air cooling techniques for substrates that will handle up to 4 watts per square cm.

5. MODE OF OPERATION

In general, the work at MCC is primarily done in-house, with about two thirds of the technical staff being hired directly by MCC, and the other third coming on assignment from the shareholder companies.

The strategy is to take development to the point of immediate usefulness to the shareholders, in a form that is ready to be used by the shareholder companies in their products. The degree of completeness necessary will vary, depending on the nature of the results, and might go as far as the construction of prototype systems. MCC will not manufacture products. The companies carry forward to the marketplace whatever products they choose to develop from the technologies given to them.

Already many results of value have begun to flow out to the shareholders. As expected, there have been spin-offs from the long-range work. A milestone was passed last summer with the announcement of the first commercial product to hit the market based primarily on MCC technology, the NCR Design Advisor, which is an expert system that assists the designer of an integrated circuit.

Although different sets of companies participate in each of the research programs, we are finding more and more opportunities for mutual benefit through collaboration and exchange of ideas on issues in common among the programs. We are finding that the number of such common issues is increasing rather than decreasing as the programs mature. The ability to take advantage of the co-location of other research programs as we have, has turned out to be a significant benefit, and is expected to result in even more productive results for the participants in the coming times.

6. THE ROAD AHEAD

In the long run, the major products from MCC to its shareholders will be ideas. There will be some designs, and some software that can be taken and used as is, but for the most part, the real value that MCC provides will be measured by the extent to which designers and developers take and apply the ideas and techniques that emerge. Success will occur only if these people somehow do their jobs differently as a result of our work. There has to be impact. To have this impact, it is not enough just to have ideas and write papers about them. One can not figuratively dump a basket of ideas on a customer's desk and expect him to do the rest of the work to figure out which ones are good, and which suit his needs.

The good ones need to be separated from the not so good, with the reasoning or experimental evidence supporting this conclusion, sufficient to satisfy and convince a designer in a parent company that he understands when he can and can not employ them with confidence. What will it take to do this convincing? It will vary. But it clearly will take results from experimental work.

Today, work in computer science tends to take the form of a demonstration. Someone builds a system to do something, (whether expert system, data base management system, parallel computer, or whatever) and then talks about it. There is nothing wrong with this, as long as the investigator keeps his scientific objectivity, reporting on the performance limitations and constraints that he finds, as well as the successes.

The more useful papers report on how well the system worked, while less useful are the talks that only focus on how the researcher built it. Worse yet are the talks about how it would have been done if there had been enough resources.

But having done a successful demonstration of some system, there is still additional information that is needed for a system architect to confidently incorporate any new ideas into designs of real systems for real paying customers. He needs analysis, or more likely, experimental data -- results that give him confidence that the ideas or approaches will work for his problem, and that he can predict with some accuracy how well they will work, and what the cost and development effort will be.

Among the things the architect needs to know about the techniques he might employ are their applicability, performance, and extendability. Under applicability fall questions like:

> What classes of problems are right for these techniques?
>
> Under what circumstances do which ones work? Work well?
>
> How do you determine or define "well"?
>
> What are the performance limitations?

Where does the approach break down?

Under what circumstances does it fail?

This last question is particularly important. If a researcher cannot tell you when it will fail, as well as when it will succeed, he has ceased to be a scientist and has become a salesman. To say that a system has never been found to fail, as I have actually heard stated once, is not praise for the system, it is an indictment of the researcher.

There is a tendency not to report negative results, yet these are also valuable for the lessons learned, and should be encouraged. Under the topic of performance fall the following class of questions:

How can the performance of a proposed application be predicted?

How does one measure the performance of a system?

What are the appropriate measurables?

How does one quantify and measure the functionality being offered? Speed? Response time? Cost? Resources needed?

Under the topic of extendability, questions arise from the fact that almost always, the demonstration has not been done at full scale. Given that something works on a small size, or for a limited functionality, how should one expect it to scale up?

How do required computing resources such as cpu speed or memory size scale with performance?

How does implementation effort scale (e.g. collecting and encoding the rules for an expert system)?

Finally, the architect also needs verification/validation techniques. How can he be sure when his system is correct and error free? With conventional software techniques this is still mathematically an unsolved problem, although many good methodologies have emerged. But for some of the newer programming techniques, such as logic programming or expert systems, the situation is not as good. Today one puts together the rules, runs the system, observes it do something wrong, examines the rules, sees why it did it, changes the rules, and repeats. After a number of iterations, the time to failure gets longer, until the builder finally declares that the wait was long enough that it must be right. Such systems can be very useful as advisors, as a tool to call a human's attention to something. But I'd rather not fly in a spacecraft controlled by an autonomous decision making system of today's level of technology.

To answer these questions will take experimentation, because theory is in no shape to predict them analytically. These experiments may

be demonstrations, but need not be. But in any case, they will have to be designed in advance to reveal and collect the desired data. At MCC we have the opportunity to do this.

To get good experimental data requires good methodology. This must be developed -- the experimental techniques that will be generally accepted as valid, and the tools for executing the experiments. There is opportunity for significant contribution here from university and research groups throughout the community to resolve what will be generally accepted techniques.

It will be a non-trivial problem, but it is time to address it for computer science. To be a true science, one must be predictive and quantitative. It must describe phenomena that are repeatable. Furthermore, today, not only is computer science an experimental science, but as such, we are still at a scale that is equivalent to test tubes and bunsen burners on a lab bench.

There are not yet big science equivalents of the high energy particle accelerators at CERN, Brookhaven and Fermilab, or of the huge radio astronomy antennas and arrays? There have perhaps been a few attempts, for example the Illiac 4, but not many.

MCC is beginning to move into this realm, as we are able to focus significant teams on specific big problems. There is now an opportunity to explore some issues that simply require large team efforts to obtain the desired data. But it is not just for MCC, but rather all of Computer Science needs to address big science experiments as well as small. There is going to be big computing. People are today building large software and distributed hardware systems. But today it is done in much the same way that the Renaissance architects did the cathedral of Notre Dame -- with intuition (even genius), and insights gained from experience, but without good quantitative theory to guide them.

It will not be easy. I don't offer you any sure fire recipe. But here also, we may see a way shown by the plans and visions of Moto-Oka and his co-workers, in their plans to build systems to demonstrate the concepts. We have seen already such systems come out of the ICOT efforts. This is the right direction, toward experimentation on a large scale. It is a direction that must be included in the whole of computer science for it to move ahead as a complete scientific field.

Since some of the experimental data must be "big science" in nature, we must develop the big science methodologies to design experiments so that meaningful data are collected, and to persuade those who allocate the resources that these experiments must be carried out.

7. CONCLUSION

A new model for performing research in a cooperative way has been created. It is an experiment, and while many questions have now been answered, there still remain some.

When MCC was formed, there were questions about whether cooperation among otherwise competing companies could be viable, whether such an institute could attract a top quality staff, whether it could produce good research in such an environment, and whether it could successfully transfer its results to its owners. The first three of these questions have already been answered in the affirmative, and we are now moving hard on the fourth. Results of value are flowing, and many technology transfer techniques are being explored to see what are the most effective ways to put the research results into productive use.

MCC's programs and plans are developed in concert with the shareholder companies, and are driven by what directions would be the best for these companies needs. Although we follow with interest the developments we can see in other research programs, including in Europe and Japan, it is not reactive to these programs. While it was stimulated in large measure by the ambitious Japanese programs and aggressive goals they so boldly set, and the feeling that in order to remain competitive in these industries, U. S. companies had to do research work to match, it is now clear that what they have is a good thing, and would continue to be so even these other programs were to disappear tomorrow.

With a consortium, the shareholder companies get something jointly they could not get individually -- the fundamental research and exploratory development that will form the base for their products of the next decade. This is possible through achieving a critical mass and economies of sharing and of scale, and because the concentration of talent that results will permit recruiting other top talent that could not be attracted otherwise.

It has thus now been shown that this concept of cooperative research can be made to work, and will provide significant benefits to its participants.

Policy, Program and Perspectives of CWI, Amsterdam

by

P.C. Baayen

Scientific Director CWI

Introduction

In this short paper we shall give an introduction to the Centrum voor Wiskunde en Informatica (CWI), to its programme and to its position within the Dutch 'Informatica Stimuleringsplan' (INSP), the governmental programme for promotion of information technology in the Netherlands. The discussion about this position is a very actual one, since the INSP will terminate at the end of 1988 and the Dutch government does not intend to extend its duration. The problem of guaranteeing continuity is of utmost importance for all institutes which benefit or have benefitted from temporary sponsoring. We feel that the problems encountered so far in this process may be of value for more general discussion.

Acknowledgement: valuable help for the preparation of this paper was provided by J.Bus and H.Nieland.

Centrum voor Wiskunde en Informatica (CWI)

General Introduction

The Stichting Mathematisch Centrum (SMC), founded in 1946, aims at 'fostering the systematic pursuit of pure and applied mathematics in the Netherlands'. From the beginning SMC kept an institute which up to 1983 was called 'The Mathematical Centre' and since then bears the name 'Centrum voor Wiskunde en Informatica' (CWI) (which means: 'Centre for Mathematics and Computer Science'). The institute was founded in a period of post-war reconstruction and its founders were convinced about the importance of mathematics for society. The Centre traditionally focusses on fundamental (pure scientific and strategic) research in mathematics and computer science.

Besides a modest attention to pure mathematics, the main emphasis lies on application-oriented mathematical subdisciplines (applied mathematics or mathematical physics, mathematical statistics, operations research, numerical analysis and, before it was recognized as an independent scientific discipline, computer science).

During the first fifteen years important activities included: the design and construction of electronic computers (the first Dutch computer was built at the Centre) and modelling and calculations in connection with the Delta Sea Defense Plan, drawn up by the government after the disastrous storm surge in 1953. The latter was an excellent example of interdisciplinary research (statistics, numerical and applied mathematics, and operations research). In the sixties attention shifted to more abstract research, in particular in the field of programming languages. The Centre played for instance an important role in developing the languages ALGOL60 and ALGOL68.

Today the main objectives of CWI still fit in the tradition of a centre for fundamental research with a responsibility to make research results applicable for solving problems in society. The main goals are:

- Carrying out *excellent research* in mathematics and computer science, primarily focussed on fundamental (pure and application-oriented) research.
- *Sharing knowledge* with and *training of researchers* for universities, industries and government agencies.
- Positioning CWI as an international *meeting point* for researchers.

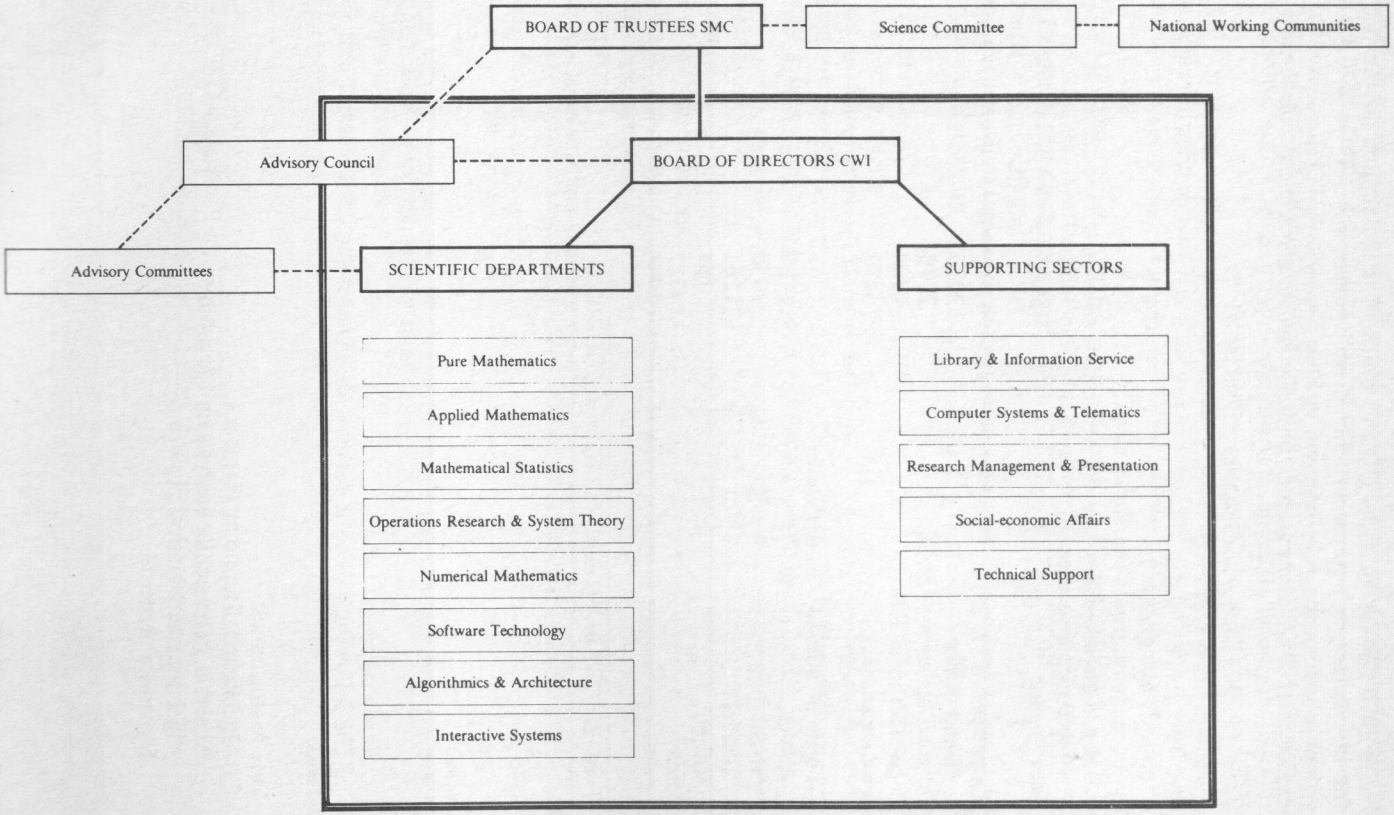
Details about the scientific programme will be given in a following section. Sharing of knowledge and training of researchers is performed on the one hand by the usual means in the academic world: publications, lectures, guiding young graduates etc., on the other hand by cooperation with industry and government agencies in strategic research projects and by research in commission. Finally, as a meeting point CWI invites researchers to work and lecture at CWI for shorter or longer terms, it organizes conferences, courses, workshops etc. For instance, Frontiers in Information Technology, the framework organization for advanced conferences in Information Technology works under the auspices of CWI. The present conference is one outcome of FIT's activities.

Organization and resources

The organizational structure of CWI is given in the organigram below (fig.1). The scientific departments are rather small units, of 10 to 25 researchers. Each department counts three to five project groups. Advisory committees make recommendations to the boards or to the department leaders on implementing research plans.

The number of staff researchers employed in 1986 at CWI equals 104. The number of graduates with four year terms aiming at a doctoral thesis is 31. Moreover, 82 people work in the supporting sectors, including the system programmers which acquire and maintain the computer equipment and its accessory software, as well as develop new software. Finally, about 10 employees of other organizations and industries carry out their research at CWI.

Fig.1: Organizational structure of CWI



In 1986 CWI spent about Dfl 16.5 million for exploitation and Dfl 1.6 million for investment in equipment. The main sponsor of CWI is the Netherlands Organization for the Advancement of Pure Research ZWO. Its share in the total exploitation of CWI amounted to 68%. Details are given in the tables below.

Table I: CWI Finances (all figures x Dfl.1000)

Income	
subsidy and grants	14387
proceeds from services and courses	1517
sales of publications	132
miscellaneous income	488
total income	*16524*
Expenses	
labour costs	12035
materials and overhead	2678
transfer to SARA	1700
total expenses	*16413*
Origin of subsidy and grants	
ZWO	11189
INSP	1750
ESPRIT	1261
other	187
total	*14387*

Table II: CWI investments (all figures x Dfl.1000)

IAS	750
ZWO	600
INSP	250
total	*1600*

Research programme

The following scientific projects are given in the CWI research programme for 1987.

Pure Mathematics:
- Algebra and discrete mathematics
- Analysis, in particular analysis on Lie groups
- Algebraic mathematical physics
- Dynamical systems

Applied Mathematics:
- Dynamical systems with stochastic perturbations
- Asymptotics
- Nonlinear analysis and biomathematics
- Analysis and reconstruction of images (see also Mathematical Statistics)

Mathematical Statistics:
- Semiparametric statistics
- Stochastic processes
- Applied statistics
- Analysis and reconstruction of images (see also Applied Mathematics)

Operations Research and System theory:
- Combinatorial optimization
- Analysis and control of information flows in networks
- System and control theory

Numerical Mathematics:
- Discretization of initial value problems
- Multigrid methods for boundary value problems
- Computational number theory
- Numerical software: (i) Ada; (ii) Vector and parallel computers

Software Technology:
- Concurrency
- Formal specification methods
- Extensible programming environments
- Interactive text processing
- Expert systems
- Logical aspects of artificial intelligence

Algorithmics and Architecture:
- Complexity and algorithms
- Architectural transparency
- Computer systems and ergonomics
- Distributed adaptive information systems
- Constructive algorithmics

Interactive Systems:
- Computer graphics
- Interactive workstations
- Dialogue programming
- Intelligent CAD systems

CWI and INSP: the Dutch Information Technology Promotion Plan

In 1984 the Dutch government launched the Informatica-Stimulerings Plan, INSP (Information Technology Promotion Plan), financed by the Ministeries of Education & Science, Economic Affairs and Agriculture & Fisheries.

The INSP aimed at stimulation of the broad field of information technology. It involves education, research, public and business information, market sector and government administration. Its duration is five years (1984-1988). Sponsoring of research involves industrial research, strategic research and fundamental research. A temporary organisation: Stimulation Projectteam for Information Technology Research (SPIN) has been set up to organize and sponsor cooperative academic-industrial strategic research. Its total budget for five years amounts to Dfl. 70 million. Besides this, the INSP provided CWI with the potential to grow into a 'leading centre for fundamental and application-oriented research into computer science'. Based on a Computer Science Development Plan, which CWI produced in 1984, a five year subsidy of Dfl. 2 million per year has been granted.

In accordance with the general objectives as given above, the strategic goals of the INSP and the special strengths of CWI (in particular its possibility of exploiting the synergy between mathematics and computer science research) in the Development Plan the following priorities in computer science and related research were chosen:

I *Algorithmics*
- Distributed algorithms
- Interactive planning methods
- Vector algorithms for supercomputers

II *Software technology*

III *Distributed and interactive systems*
- Interactive systems
- Distributed operating systems
- Performance analysis of computer systems
- Distributed information systems

IV *Pattern recognition and artificial intelligence*
- Statistical image analysis
- Expert systems and other aspects of artificial intelligence

The choice of theme I was based on its key role in computer science, as well as the possibility of synergy with mathematical research at CWI. Theme II was for many years already a research priority at CWI. The main part of this research however has been financed from other funds than INSP. Theme III mainly concerned extension of existing and building-up of new expertise in application-oriented research. Finally, in theme IV mainly new research of great strategic importance was initiated.

Moreover, measures were proposed to strengthen the knowledge transfer and the position as a meeting point.

CWI has been able to increase the research and supporting capacity with the INSP-subsidy according to the table given below. Moreover, part of the INSP-money is used for acquiring equipment, as to improve the rather poor computer infrastucture of CWI.

Table III: Distribution INSP subsidy (all figures x Dfl.1000)

	1984	1985	1986	*total*
Awarded	2000	2000	2000	*6000*
Investments	2000	900	250	*3150*
Exploitation	-	1100	1750	*2850*

CWI and industry

CWI's relations with industry go back to the construction of the first computers in The Netherlands in the fifties. This case already exhibited a principle of policy which CWI still maintains: transfer of expertise to another organization or institute, as soon as research develops to a stage (demonstration project or prototype) beyond which it no longer fits in with the Centre's function as institute for fundamental research. Thus in 1956 N.V.Elektrologica was set up to carry on computer construction work.

Throughout its existence the Centre has acted in a consultancy capacity, e.g. for industry. Recent years have brought an increase in the number of joint projects with industry on a national and European level. At home, such projects involve the Foundation for Technical Sciences (STW) and SPIN, and at the European level ESPRIT.

Joint research projects with industry are an especially attractive means of knowledge transfer. In the case of STW projects, industry is involved as potential user and has a place in the user's committee. Other joint projects come under ESPRIT, SPIN or other government enterprise schemes. Others again are set up with major financial support from industry. Lastly, there are projects involving researchers detached by their companies to carry out work at CWI (a possibility which until now has not been sufficiently exploited). In 1986 joint research accounted for almost 20% of total income. CWI policy is directed at increasing that share as far as possible. In table IV we give a short summary of international and national cooperative projects.

Table IV: CWI participation in (inter)national programmes

European programmes

ESPRIT

METEOR (432): An integrated formal approach to industrial software development
GIPE (348): Generation of Interactive Programming Environments
415: Parallel Architectures and Language for AIP - a VLSI directed approach
DIAMOND (1072): Development and integration of accurate mathematical operations in numerical data processing
VIP (1229 (1283)): VDM Interfaces for PCTE

Other programmes

European Commission Project: Pilot implementation of basic modules for large portable numerical libraries in Ada
COST-11: Distributed Systems Management
ESA project HERMES: Convergence acceleration of a finite volume Euler solution with a geometric and algebraic adapted multigrid method in finite volume Euler solutions

National programmes

SPIN (Stimulation Project Team Computer Science)

PRISMA: Parallel Inference and Storage Machine
FLAIR: Flexible Automation

NFI (National Facility Computer Science)

Decision support systems for operational planning
REX: Research and Education in Concurrent Systems
Transformational programming
Cryptography and Computer Security
Intelligent CAD systems

STW (Foundation for the Technical Sciences)

Facilities for raster graphics in programming languages
Shallow-water equations: evaluation and stabilization of numerical methods
Mathematical methods for the analysis of spectral atmospheric models
Development of efficient numerical methods for flows described by the stationary Euler equations
Statistical analysis of traffic flows
Prediction and control problems for the Dutch freeway control and signalling system
Overload control of communication systems
Adaptive grid techniques in software evolutionary partial differential equations
New architecture for interactive raster graphics on the basis of VLSI

IOP (Innovative Research Programmes)

IC-Technology: Numerical Methods for Semiconductor Device Modelling
ISNAS: An Information system for simulation of flows with the Navier Stokes Equation

An important way of industrial involvement consists of a company financing research fellowships at CWI, in a jointly selected area of fundamental research, for a number of years. Such industrial fellowships contribute to CWI research and the fellows are supervised by CWI project leaders: at the end of the contract the researcher and the knowledge he has gained, are available to potential employers. Shell created two such fellowships in 1986, and it is expected that other companies will follow suit.

It is CWI policy that suitable research findings, e.g. such software packages as GKS-C and CAR (Computer Aided Routing), achieve their commercial potential. Whenever possible CWI cooperates with established concerns prepared to develop, market and sell the software under licence in return for royalty rights on income. When this is not feasible, consideration is given to participation in new companies, set up with CWI as base, to market the fruits of research. This can mean both consultancy in the fields of mathematics and/or computer science, or the development and marketing of software products.

CWI after the INSP-stimulation

It requires a permanently higher level of sponsoring after the INSP-period to maintain at least the position built up during that period in computer science research. CWI expects to acquire the necessary funds from (i) substantial and permanent accretion by CWI's main sponsor ZWO, (ii) permanent project sponsoring by government agencies, and (iii) increasing income from commissions,

In tables V, VI and VII we give an overview of CWI finances in recent years. Table V shows the proportion of total operating costs for the years 1981-1986 covered by the ZWO grant. It is clear that an increasing proportion of finance comes from sources other than ZWO. Although the INSP grant accounts for a proportion of these funds during 1985 and 1986, the further breakdown of income in Table VI shows that other forms of research subsidy and commission fees account for a growing contribution. This source of growing income is particularly related to research into applied computer science. A continuation of this upward trend is foreseen in 1988 in view of involvement in new EEC projects (reinforcement of CWI research into software technology from 1985 is mainly realized by participation in ESPRIT programmes), SPIN projects and innovative research programmes subsidized by the Dutch government. Finally, Table VII summarizes sources of finance for equipment purchase; as is shown, the ZWO share has fallen from 100% to just over half. However, almost all the remaining amounts concern temporary enterprise grants.

Table V: CWI-exploitation 1981-1986

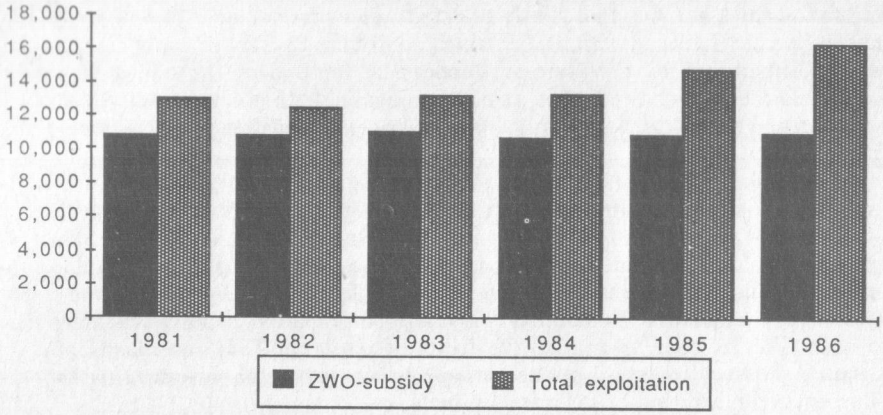


Table VI: Breakdown of non-ZWO income

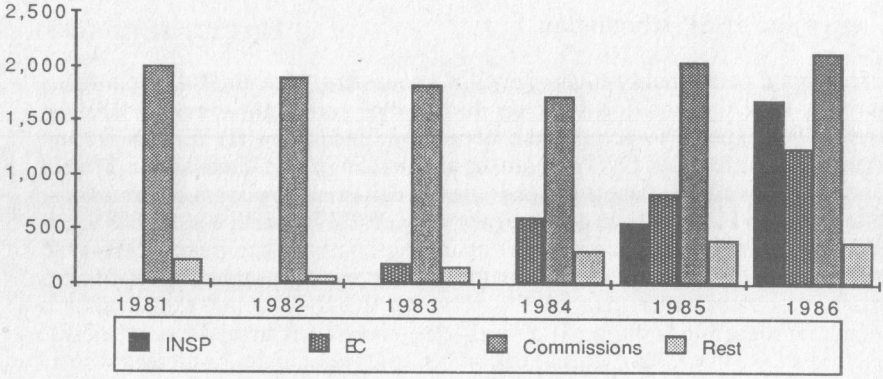
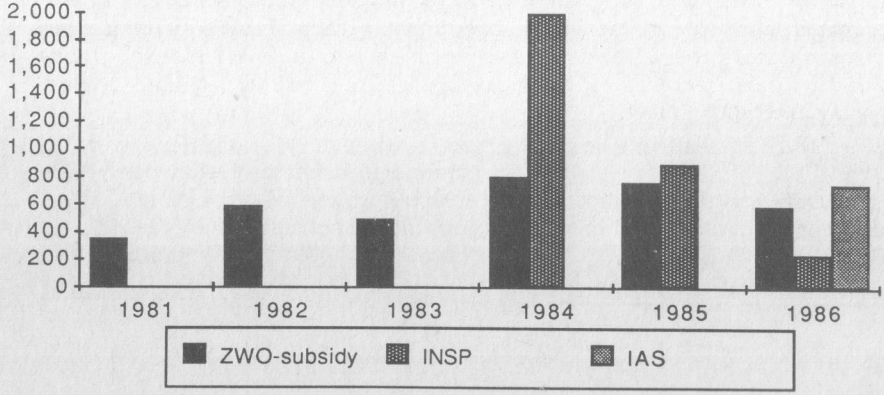


Table VII: CWI-investments



It may be concluded that the main goals set for the INSP-sponsoring period have been achieved, within the limits of the money acquired. It is also clear that the formula worked. A substantial sponsoring of fundamental research by government funds (ZWO and others) leads to a firm basis of expertise which can be used in cooperative strategic research projects. This yields an excellent support of precompetitive industrial research and industrial development. It is CWI's conviction that a permanent ratio of about two third government subsidy for fundamental (free) research and one third to be obtained in strategic research, commission and other income gives a stable balance and a sound position for a research institute as CWI is.

This yields two conclusions:

1. The basic subsidy, which has been increased on a temporary basis, should be provided on a permanent basis by the science organizations. This means increase of the budget of these organizations. (Although reallocation is a theoretical possibility, the increasing dependence of society on science justifies an increasing financial commitment of society.)
2. Sponsoring of strategic research, on a basis of partial payment so that industry and institute each bear a part of the responsibility, should be built in as a structural possibility in our government research organizations. The need of shorthening the path from theoretical research to product development makes such sponsoring indispensable; the clear success gained in the last stimulation period proves that it can be highly effective.

ECRC: a Joint Industrial Research Centre

Hervé Gallaire

ECRC
Arabellastr 17
D 8000 Munich 81- FRG

This paper describes the policies, strategies and mechanisms of ECRC, as well as its aims and goals. It also surveys its technical program. The paper is non-technical by nature, the emphasis being put on the rationale for such a centre, and on its operations.

1. CONTEXT AND AIMS

The ECRC (European Computer-Industry Research Centre) was created through the enterprise of three major European computer manufacturers: Bull, ICL, Siemens. Its activities are intended to enhance the future competitive ability of the European Information Technology industry and thus to complement the work of national and international bodies. ECRC started its operations on 1st Jan, 1984.

It is a commonplace to assert that information technology (IT) will play an increasingly important role in the world of tomorrow.

The chief difference between today's computers and those of tomorrow lies in the ability of future computers to represent, memorize and process not only numbers but also symbols without numeric interpretation. When symbolic information is stored today, it is essentially used for retrieval purposes only. Symbolic processing and the use of the methods of artificial intelligence will enable the computers of the future to represent knowledge, to simulate and reproduce reasoning patterns on the basis of "inference" systems.

The basic knowledge to be represented in symbolic form as well as the knowledge required to interpret and combine it have to be introduced into the system, however. This will lead to new programming languages and methods, and to new forms of software. Through the introduction of very high-level languages a far-reaching revolution in programming techniques is already underway.

The desired levels of performance of programs written in these languages will also require processing speeds and capacities much greater than those of today's systems. Because of this, system architectures are also undergoing radical changes, and, in step with these changes, parallel processing methods are being introduced which can in part be achieved using existing technology.

Finally, the user will have to be able to interact directly with the computer, which

must then provide responses, clearly stated and fully justified.

Thus the computer of tomorrow will offer users much more than simple number-crunching, providing instead truly deductive and interactive assistance. These systems will find a vast range of applications and they will be of particular benefit in the important area of decision-making. Hence the emphasis on this particular topic at ECRC.

The aim of ECRC is to develop fundamental know-how. The Centre's field of activity includes technologies needed to improve the process of **computer-assisted decision-making**.

The program encompasses the set of techniques - and the necessary software tools or machine architectures to support them - needed to resolve the issues of

- knowledge description and acquisition, where knowledge comprises both basic information and situation-specific decision rules;
- reasoning mechanisms which build on the aforementioned knowledge;
- interaction models and tools to enlarge the involvement of both the user and the computer in the decision-making process.

2. TECHNICAL PROGRAM

2.1. Global Strategy

The field of Computer-Assisted Decision-Making (CADM) relies on specific techniques such as optimization and operations research techniques, databases, and on a wider field, Artificial Intelligence (AI). What characterizes the CADM field today is the use of AI tools.

A Computer-Assisted Decision-Making system must combine in an efficient and human friendly way the techniques proposed by AI, such as knowledge bases, expert systems, with conventional techniques such as databases. These are some of the challenges involved in building such systems. Progress in the field will thus be reached through a dual approach:

- research and development of solutions to problems which have been identified as bottlenecks for CADM systems building, and
- integration of already-developed techniques in prototypes satisfying requirements at application or systems level.

ECRC is thus engaged in a strategy based on two parallel lines of work, one dealing essentially with independent studies in technology improvement aimed at providing the building blocks of the domain (methods, languages, basic tools, methodologies), the other aiming at assessing the effectiveness of the proposed technology in constructing adequate research prototypes. These two lines are being followed concurrently. Improving the decision-making process will require an

extension of our knowledge in a number of technical spheres.

2.2. Program Outline

In order to concentrate its resources and develop internal consistency, the research program is structured around two key concepts:

- the fundamental role of knowledge bases in developing systems to assist decision-making,
- the importance of logic programming in defining and implementing concepts, although on a non-exclusive basis.

Knowledge Bases will lie at the heart of most software systems, systems to which they bring power and flexibility. To provide the functionalities they promise, these systems require the existence of various tools to represent and manipulate information as well as to reason and interface with it. A Knowledge Base Management System (KBMS) will be a target of ECRC.

Logic Programming is a programming paradigm which provides a good base for what is needed in building and using these systems, due to its dual knowledge representation and inferencing capabilities. **Problem solving techniques** are necessary in applications, in particular in specific knowledge-based systems built from a KBMS; they will be studied here to a large extent in the logic programming framework. Research is needed to develop basic problem-solving techniques with wider applicability. Logic Programming can also be used as a language out of the scope of knowledge bases, and research in this direction will be carried out.

Interacting with the complex systems necessary for assisting the decison-making process, in particular Knowledge Base Management Systems or their applications, poses a number of challenging problems which will be studied. This interaction involves issues of presentation of the knowledge used by a KBMS, of exploration of an application by their users; it depends on the type of user and on the application. Human factor studies are necessary in this framework. Work in this area clearly also involves mastering the building of interactive systems.

The processing requirements of those decision support systems which provide symbolic reasoning mechanisms to exploit existing knowledge will be examined, and **architectural** issues aiming at providing the kind of speed and environment required to make these systems attractive will be researched.

ECRC projects maintain the continuity with those basic tools closely related to standard technology - e.g. databases - as used by the shareholders (the three member companies of ECRC; see section 3).

2.3. Structure of the Research Program

Consequently, the program of ECRC is currently centered around four areas, the contents of which is broadly described as follows:

- **Logic Programming and Problem Solving Techniques**
  Implementation and efficiency issues: integration of more powerful programming and problem solving styles including, for example, object-oriented programming and constraints propagation techniques; programming environments tools including debugging aids; incremental programming techniques: exploration and explanations of decision systems.

- **Knowledge Bases**
  Extensions of the relational model used in database systems to include deduction capabilities and semantic capabilities to deal with a richer description of the world, architectural issues involved in putting together the various components of a KB system, theoretical issues arising in such systems, e.g. consistency, integrity checking, completeness of search.

- **Computer Architecture**
  Symbolic computing architectures adapted to the execution of the above systems including sequential and parallel systems, design of specific architectures such as coprocessors, modelling of parallel computational models.

- **Human Computer Interaction**
  Investigation of the issues of presentation, exploration of a CADM system, study of the techniques necessary to specify and develop graphic interfaces in a more systematic way and tailored to specific applications or users.

The work is organized so as to allow projects to cross over the borders of the four areas. The technology studies are well underway in various areas. This has allowed early research prototype development and some application simulations. Results of some studies have been transferred to the shareholders: a PROLOG compiler, a knowledge base prototype, a technique for graphic interface development, a language for problem-solving and programming. More are scheduled, in particular a very fast sequential inference machine.

2.4. Applications

ECRC is not devoted to specific applications development, and its work is more technology driven than application driven. Obviously, however, ECRC uses the analysis of application domains or of applications to try to comprehend the capabilities of the technology it is investigating as well as its limits. For example, ECRC is looking in depth at applications in the finance and economic fields, to analyze how decision making tools could benefit from explanation techniques. Similarly ECRC has spent significant resources to investigate the domains of combinatorial optimization as well as electronic circuits design and automation tools in order to make sure that the work done on problem solving using constraint propagation techniques was not valid for toy applications and puzzles only.

3. How does ECRC work

3.1. Roots of ECRC

ECRC is an industrial research centre in Munich supported as a common resource by the major European computer manufacturers. Bull. ICL and Siemens. It is owned equally by the three and they share equally its cost. This is an original feature of ECRC. which makes it different from, e.g. MCC. Clearly with only three partners this was the best way to ensure collaborations. ECRC is an independent company operating under German law (GmbH). The creation of ECRC reflects current industrial European thinking, directed to consolidating Europe's presence in the electronic data processing market: scientific cooperation, standardization, implementation of technical standards, etc. It parallels current European initiatives, e.g. the EUREKA project and the European Community's ESPRIT program, agreements for standards between industries or between the EEC member states. But ECRC is not supported by governments or European programs. This makes it different from, e.g. ICOT. Up to now, ECRC has not been a contractor in European projects either.

3.2. Organization

ECRC is organized in the following manner:

- ECRC management;
- the Shareholders' Council which approves programs and budgets and monitors their execution;
- the Scientific Committee which advises the Shareholders' Council in determining future research directions.

3.3. Policy

Research is carried out on behalf of the three member companies, which use the results freely, all rights in the results of research being shared by all three companies who have a free license on all patented results of ECRC.

As mentioned above, ECRC is an industrial research centre. What does this mean? ECRC is not a development centre, and there is no commercial agreement between the three to exploit particular aspects of ECRC or to fix the research program of ECRC based on a product target. ECRC. as any research centre, engages in whatever theoretical and/or applied research appears necessary to solve the scientific and technical problems that arise while trying to achieve its aims. As examples, ECRC has been doing theoretical research on theorem-proving techniques for checking the consistency of sets of rules, on problem-solving techniques based on constraint propagation mechanisms, on automatic evaluation of recursive queries to a database, on improved mechanisms for the evaluation of negation in logic programming, or on computability theory. This research has been widely published. Less theoretical research has been conducted in other fields as described in the outline of the program. To be an industrial centre means that ECRC has to evaluate the potential of its results: how feasible are the solutions arrived at? Will they lead to toy or realistic systems? What benefit can they bring to ECRC's shareholders? Obviously this requires analysis, research prototype developments, experimentation with them and a kind of feedback loop to take place. A balance

between these different types of activities is difficult to find. Thanks to the temporary assignment of a significant part of its staff, ECRC is perhaps in a better position to find and preserve such a balance.

3.4. Personnel

The personnel structure of ECRC is made up of several categories:

- research staff - there are two types:
  - researchers detached by the shareholders to ECRC. These may be personnel having worked in the companies or hired by the companies to be immediately detached to ECRC. They form about 70% of the research staff. Their assignment is for an initial 3-year period which can be extended
  - researchers directly hired by ECRC, without time limitation
- administrative staff, including a technical group. These employees are essentially permanent
- collaborators: these are different categories of researchers, including full-time researchers from public research centres detached or on sabbaticals, doctoral students, part-time researchers, consultants, etc. They have contracts of limited length.

ECRC currently has 16 different nationalities among its staff of 70. The average age of the research staff (50 full-time employees) is around 30, and about 50% of them hold doctorates.

3.5. Transferring Results

The personnel structure of ECRC has been thought of as one of several means for transferring results back to the companies. The transfer of knowledge and results is one of the major problems that have been addressed at the time ECRC was created and on which we are still working. This problem is not specific to ECRC and is indeed a concern for all research centres. One can only imagine that an independent company should have more difficulties than an in-house research centre.

- First the problems: there are the usual ones, due to a long time lack of understanding between research and industry, especially in Europe. ECRC cannot be an exception. There are other problems specific to ECRC situation. For instance, can ECRC have access to marketing decisions and strategies of a particular company, or will all companies avoid giving crucial information for fear of telling too much to the other shareholders. Can ECRC establish sound relationships with the research and development units of the companies, who can either see ECRC as a competitor, when both work on similar topics (although this could be seen to be a way to maximize transfer) or ignore ECRC if the topics selected by ECRC are too different from their own concerns.
- What solutions can be found? There is no one solution, and ECRC is experimenting with its shareholders in several ways in parallel:

- staff transfer: this is an excellent way, but only a long-term one, and consequently its effects have not been very spectacular yet.

- liaison officers: each company has assigned tasks to some its staff for each area of ECRC's work to follow that work. The liaison officer should identify the right partners for ECRC within the companies: people interested in testing, using, discussing, developing results from ECRC, as well as the researchers working on related matters in case they have not been identified by ECRC. This is an excellent mechanism; its efficiency depends on who is in charge.

- seminars: ECRC organizes regularly technical seminars for its shareholders on selected topics. The motivation is to give a detailed presentation of the projects underway in a specific area and to have feedback from other researchers, developers and possibly staff in charge of strategy. As every company gets to present its work, at least to some extent, these seminars not only provide initial personal contacts or reinforce them, but are also a forum for a discussion of a whole topic. Here again results depend on attendance.

- common projects: although there will not be many such projects, it is possible for one company to work directly with ECRC on one of its projects; this will normally be on behalf of the other two companies, but not necessarily, if this was commonly agreed. There is already one such project in the architecture group, where ECRC has access to the technology of its shareholder who is developing an industrial prototype (as opposed to a research prototype).

- short period exchanges: this involves personnel of the shareholders spending an extended period at ECRC. This has been taken advantage of and looks very promising. The converse is also being considered.

The rate of exchange is definitely increasing, whether one measures it by the attendance at seminars, short stays, joint projects or informal and direct contacts. It is a positive contribution to a common understanding of the future. Transfers of results have already taken place, which have been integrated in existing products; new products based on ECRC research prototypes are now under development.

3.6. Collaborations

ECRC has not been involved in formal collaborations with other parties than its shareholders, except for some pilot users of the prototypes ECRC produced. There are several reasons to this situation: ECRC started from scratch and needed to develop its own research culture, its own program: ECRC top priority as far as collaborating is concerned, was and still is its shareholders: this has required and still requires much attention. Informal collaborations, exchanges and visits have been always positively considered, including joint workshops with academic research teams. Several such joint workshops have already taken place. The collaboration policy may be revised in the light of other factors (e.g. ESPRIT participation).

4. Conclusion

Given the time frame in which ECRC has been active, the results make us optimistic about the interest and the future of such joint ventures. Results have been obtained and published: transfer of some of them and of research prototypes has taken place, and products based on ECRC work are already appearing; more are coming. Research ventures of the ECRC type open possibilities for tighter collaborations between the partners.

Data Flow Computing and Parallel Reduction Machine

Makoto Amamiya

NTT Software Research Laboratories
3-9-11 Midoricho Musashino-shi Tokyo 180 Japan

Abstract

This paper discusses a parallel graph reduction model and its implementation in the relation to the data flow computing scheme. First, a parallel graph reduction mechanism and its relation to data flow computing are discussed. In the discussion, the data flow computing model, which introduces by-reference concept is shown to be a natural implementation of parallel graph reduction model. Then, a practical implementation of a parallel reduction machine is presented. In the implementation, a cell token flow model is proposed. By using the cell token flow concept, a given data flow program can be transformed to an efficient multi-thread control flow program. Last, an overview of the machine architecture is described.

1. INTRODUCTION

Recently, several machines are proposed for both serial and parallel reductions [1-9]. Almost all of serial reduction machines are based on the stack control mechanism, and are much more practical than those for currently considered parallel reduction machines. The most serious obstacle to the current implementation of the parallel reduction machine is that they lack in efficient parallel reduction control.

The data flow computing scheme offers a basis for parallel execution control in the most natural way [10,11,12]. Data flow based computation control can take advantage of parallelism quite naturally, since all operations are controlled according to data dependency relation. Programmers can describe the precise synchronization control using data flow concept.

Control in the reduction model lies in the choice of reduction strategy, i.e. the order of reduction. The best known reduction strategies are applicative order reduction and normal order reduction. In general, applicative order reduction is more efficient, while normal order reduction is safe for nonstrict evaluation. Therefore, it is suggested that the normal order reduction should be used only in a few parts where nonstrict evaluation is needed, i.e. lazy evaluation should be performed only where it is necessary.

This paper discusses the relationship between reduction and data flow computing, and proposes a model and implementation of a parallel graph reduction machine based on the data flow computing scheme. It is shown that the conventional by-value data flow computing scheme implements the applicative order reduction strategy. In the proposed model, the data flow computing scheme is extended so as to implement the normal order reduction, by introducing by-reference concept. The extended data flow computing model carries the data flow graph itself, which is called meta-data. It is also pointed out that the data flow computing model naturally implements the supercombinator [13].

Then, the paper discusses a parallel reduction machine implementation. The machine architecture is designed on a multi-thread control flow concept. The multi-thread control flow program, which is extracted from the given data flow program, reflects the data flow structure of the given program, and yet eliminates the need for redundant

data flow controls, such as switch and gate controls, and overhead for operand matching. This makes the data flow machine much more efficient and practical. Last, an overview of the parallel reduction machine architecture is described. The machine is designed on the basis of packet communications concept, in which many processor and memory resources are connected with packet communications network. The processor is designed as a circular pipeline system which can execute the multi-thread control flow program in high speed.

2. GRAPH REDUCTION

Essential constructs of functional programming language are expressions and functions. Execution of a functional program consists of evaluation of expressions. The evaluation of expression is interpreted elegantly as a reduction process.

Graph reduction is performed on a graph which is a representation of a given functional program. Graph reduction is stated briefly as follows. (1) Expression has a natural representation as a tree, or more generally a graph. (2) Evaluation proceeds by means of sequence of simple steps, called reduction. (3) Each reduction performs a local transformation of the graph. This process is called graph reduction. (4) Reduction for subexpression may safely take place simultaneously since they do not interfere with each other. (5) Evaluation is complete when no further reducible expressions.

Function application is also an expression. The reduction process for the function application substitutes it with its function body, in which bound variables are substituted with actual values. Then the reduction is performed to the function body.

We will inspect the graph reduction process using a simple example. Expression (5+3)*(5-2) is represented as a tree, and the graph reduction on this expression is depicted as Fig.1.

$$(5+3)*(5-2) => 8*3 => 24,$$

As an example of function application, when the function f

$$f = \lambda(x,y)[(x+y)*(x-y)]$$

is applied to (5,2), then reduction proceeds as

$$f(5,2) => (5+2)*(5-2) => 7*3 => 21.$$

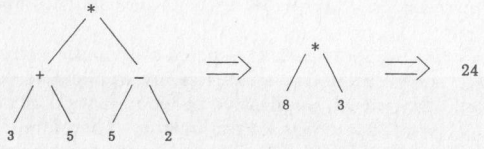


Fig.1 Graph Reduction Process

2.1 Applicative Order versus Normal Order

There are several reduction strategies. In those, normal order reduction and applicative order reduction are typical. The normal order reduction tries to reduces an expression from outer to inner, or from top to bottom in the graph reduction. The applicative order reduction, on the other hand, reduces an expression from inner to outer, or from bottom to top in the graph reduction. For example, normal order reduction of the expression (5+3)*(5-2) tries first to reduce the expression (...)*(...). Then, in order to reduce this expression, the reductions are tried for expressions 5+3 and 5-2. At this point, these expressions are reducible to 8 and 3, and the outer level expression is changed to 8*3. As 8*3 is now reducible, this expression is reduced to 24. In this

reduction process, demands for reduction propagate from outer to inner (or from top to bottom) in the first phase. When the demand has reached innermost level (or bottom), the innermost expression is reduced to normal form, which is called a value. In the second phase, the reduction resumes from inner to outer (or values are transmitted from bottom to top), and final result is obtained when the outermost reduction completes. This process is shown in Fig.2.

In the applicative order reduction, on the other hand, reductions for the expression (5+3)*(5-2) are performed to the inner most expressions 5+3 and 5-2 first, then the reduction is performed to the outer level expression 8*3. This process is shown in Fig.1.

We can easily recognize that applicative order reduction is more efficient than normal order reduction, since the demand propagation in normal order reduction is not needed in applicative order reduction. Nevertheless, normal order reduction is preferable because the normal order reduction is safe for nonstrict function. Typical examples of nonstrict function are conditional and structure data manipulation.

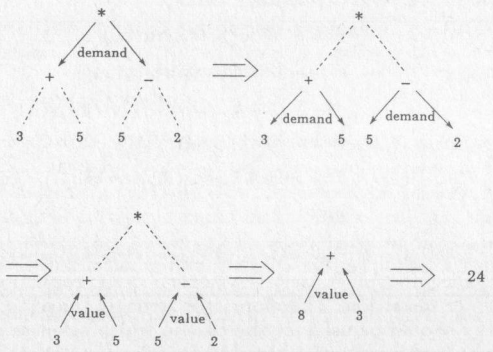


Fig.2 Normal Order Reduction (Demand Driven)

A conditional function $ite(p, e1, e2)$, which is a short form of $if\ p\ then\ e1\ else\ e2$, is defined as: If p is evaluated to $true$ then its value is the value of $e1$, else its value is the value of $e2$. In the normal order reduction, the reduction of ite is performed to p at first. Then, according to its result, the reduction is performed to $e1$ or $e2$ selectively.

If the applicative order reduction is applied to the expression $f(0,1)$ where $f = \lambda(x,y)[ite(x=0, y, f(x-1,y))]$, the reduction is performed simultaneously to expressions $x=0$, y and $f(x-1,y)$ when the reduction is performed to the body of $f(x,y)$. The last expression $f(x-1,y)$ is applied to itself recursively, and this reduction is performed to $f(x-2,y)$ again. This process continues forever.

$$
\begin{aligned}
f(0.1) &=> ite(true, 1, f(-1,1)) \\
&=> ite(true, 1, (ite(false, 1, f(-2,1))) \\
&=> ... \\
&=> ite(true, 1, (ite(false, 1, (,,, (ite(false, 1, f(-n,1)))...))) \\
&=> ...
\end{aligned}
$$

On the other hand, if the reduction of $f(0,1)$ is performed in normal order, $f(0,1)$ is reduced to 1 safely.

Another example is nonstrict data structuring operations. The function $cons(x,y)$ constructs a data with two elements x and y, and function *head* or *tail* takes out the

first or second element from the cons data. Using *cons*, *head* and *tail* functions, *intseq* and *nth* functions are defined as,

$$\begin{aligned} intseq &= \lambda(n)[cons(n, intseq(n+1))] \\ nth &= \lambda(x,n)[nth0(1,x,n)] \\ nth0 &= \lambda(i,x,n)[ite(i=n, head(x), nth0(i+1, tail(x), n))] \end{aligned}$$

When the expression $nth(intseq(1),2)$ is reduced in applicative order, the reduction is performed to $intseq(1)$ first. In the course of reduction of $intseq(1)$, the reduction is performed to $intseq(2)$ before $cons(1, intseq(2))$, and this reduction is again performed to $intseq(3)$ recursively. Thus, reductions to $intseq(n)$ is performed infinitely.

If the reduction is performed in normal order, $nth(intseq(1),2)$ is reduced to a normal form safely.

$$\begin{aligned} nth(intseq(1),2) &=> nth0(1, intseq(1), 2) \\ &==> nth0(2, tail(intseq(1)), 2) \\ &==> head(tail(intseq(1))) \\ &==> head(tail(cons(1, intseq(2)))) \\ &==> head(intseq(2)) \\ &==> head(cons(2, intseq(3))) \\ &==> 2 \end{aligned}$$

The normal order reduction is thus safe and more powerful especially for nonstrict operations. However, the problem is that the normal order reduction is less efficient than the applicative order reduction. Therefore, it is strongly suggested that applicative order reduction strategy should be used for the case in which safeness of computation is assured, while the normal order reduction should be performed only to the case in which computation will be unsafe.

It should be noted that redundant demand propagation can be eliminated at compile time by using data flow and nonstrictness analysis techniques [14,15]. Thus, almost all of the part of evaluation is performed in applicative order at run time. Normal order reduction is performed only for a few part of evaluation, in which lazy evaluation is needed.

2.2 Graph sharing

Other important concept is sharing of the reductions. In the sharing of reductions, the reduction to an expression which is shared with several expressions is performed only once when the first demand for reduction is issued. Once the expression is reduced to a normal form, re-reduction is skipped for other demands and this normal form is used for subsequent reductions.

For example, consider the variable x which is defined as

$$x = (3+5) * (5-2).$$

If the reduction to an expression $x * x + x$ is performed in straightforward, the same reduction to (3+5)*(5-2) is performed three times. But this redundancy of reduction can be avoided by sharing the result of reduction to (3+5)*(5-2) as a value of variable x. This reduction sharing is elegantly controlled in the graph reduction scheme; Initially, the variable x has a pointer to the graph of expression (3+5)*(5-2). Once a demand has triggered the reduction to x, the expression (3+5)*(5-2) is reduced to 24, and x is rewritten with this result value. For the subsequent demands delivered to x, no reductions are performed.

2.3 Supercombinator

One of the problems in reduction is the occurrence of free variables. In order to treat these free variables, a set of variable-value binding informations, called environment, needs to be conveyed during the reduction. Managing such environments causes serious overhead. However, fortunately, it is possible to eliminate such free variables from the original lambda expression, and transform to a lambda expression with no free variables, by using lambda-lifting technique [16]. Such a free variable free lambda expression is called supercombinator [13].

For example, $f = \lambda(x)[x + y * y]$, which has a free variable y, is transformed into a lambda expression with no free variables $g = \lambda(y)[\lambda(w, x)[x + w * w]y]$. This expression is rewritten as, $g = \lambda(y)[f(y, x)]$ and $f = \lambda(w, x)[x + w * w]$ (or in supercombinator form, $g\ y = f\ y\ x$ and $f\ w\ x = +\ x\ (*\ w\ w)$). When h is defined as $h = \lambda(x)[g(5)]$, the reduction proceeds as

$$h = \lambda(x)[g(5)] => \lambda(x)[f(5, x)] => \lambda(x)[x + 5 * 5] => \lambda(x)[x + 25]$$

This example shows that a partial evaluation is possible by performing the applicative order (eager) reduction.

3. DATA FLOW COMPUTING

In data flow computing, programs are represented as a directed graph called a data flow graph. The data flow graph has several input and output ports. Nodes in the data flow graph represent operations. Operations on nodes are performed when their needed operands are available, and their results trigger the succeeding operations. Execution starts when initial tokens are put into the input ports of the data flow graph, and is complete when result tokens are output from its output ports. This data flow computing process is interpreted as a reduction process performed on the data flow graph [17]. For example, graph reduction shown in Fig.1 is also understood as a reduction on the data flow graph.

We can consider several data flow computing model according to what kind of data is assigned to tokens. Primitive data flow model assigns data itself to tokens. We call the computation mechanism built on this assignment as by-value mechanism. More advanced dataflow computing model assigns to the token a cell which is variable or pointer to structure data or program code. We call the computation mechanism built on this assignment as by-reference mechanism. Even more sophisticated data flow models, those that provide more elaborate computation control, are combination of by-value and by-reference mechanism.

In the by-value mechanism there is no notion of cells that hold data value or pointer. We can easily understand that the by-value data flow computing model is equivalent to the applicative order reduction.

3.1 By-reference Mechanisms

In the by-reference mechanism, all operations have their own cells which hold their operand and result values. Each cell is shared between two operations, i.e. the cell is used as result value cell of one operation and also as the operand value cell of its successive operation.

Either of two types of implementation are considered depending on how to control the read and write access to this shared cells:

(1) Cell-driven eager computation

One implementation is called cell-driven control, in which operations are initiated when operand cell is made available. The initiated operation tries to fetch its operand and waits for its value to be written by the preceding operation. A read-ready tag is used to control such concurrent read and write accesses. A simple example of cell-driven control is shown in Fig.3.

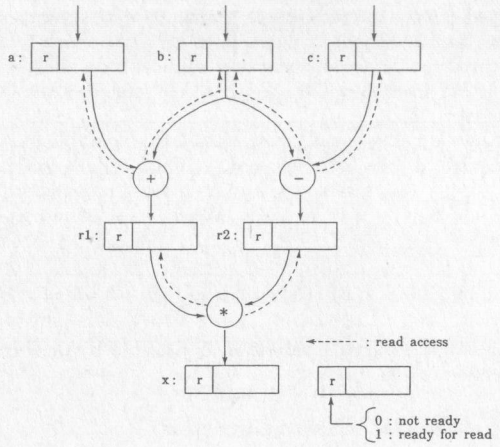


Fig.3 By-reference Mechanism (Cell Driven)

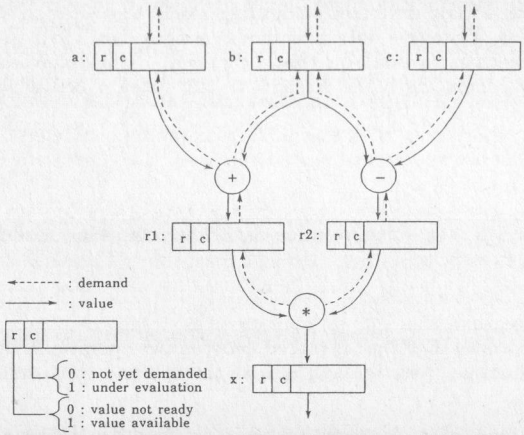


Fig.4 By-reference Mechanism (Demand Driven)

This mechanism realizes maximal eager evaluation, since all operations are initiated eagerly when their cells become available. (It should be noticed that all cells are possibly made available in advance before execution.) The cell-driven mechanism is also equivalent to the applicative order reduction.

(2) Demand-driven lazy computation

Another implementation is called demand-driven control, in which every operation is initiated when its value is required from its destined operation. In the course of execution, demand tokens flow against the direction of arc, then result values are returned back. This mechanism is shown in Fig.4.

Initially, each cell has pointer to the operation node that generates its value. When an operation is initiated, the operator accesses its operand cell. If the cell contains no value, the read access triggers a demand to the preceding operation and waits until the value arrives. In order to control this demand-triggering and value-awaiting mechanism, two tags, read-ready tag r, and demand-trigger tag c, are used in each cell.

The demand-driven computation mechanism is equivalent to the normal order reduction, and is safe for nonstrict function evaluation. However, this model has two problems in cost effectiveness. First, it is expensive to implement the demand-driven control mechanism. Second, computation takes much more time than data-driven control does because of the double flow of tokens.

3.2 Combined By-value and By-reference Mechanism

More effective mechanism of eager and lazy evaluation is to use the by-value and by-reference mechanisms selectively. This model is a data flow implementation of the reduction control which performs applicative order reduction and normal order reduction selectively. In this model, each token has as its value either a data flow graph which is called meta data, or a normal form which is called object data. Cells are allocated to variables and function argument, which are extracted by a compiler through data flow and nonstrictness analysis.

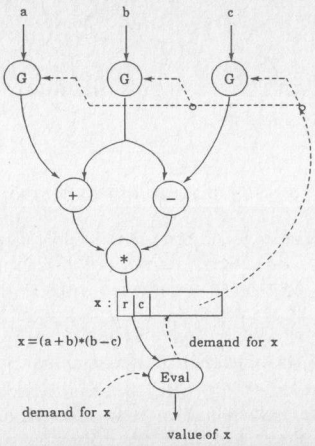


Fig.5 Combined By-value and By-reference Mechanism

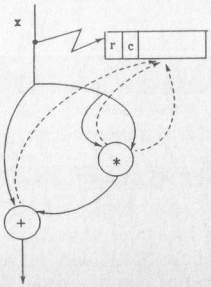


Fig.6 Graph Sharing

Each cell has two tags r and c, which are used for eager and lazy evaluation control. The content of each cell is either an evaluated value or a pointer to a program code

entry, called recipe. When tag r is on, the cell contains a value and thus read access is permitted. When tag c is on, the content of cell is recipe. A simple example is shown in Fig.5. In this graph, by-reference mechanism is used for variable x. When the value of x is required, a demand signal is triggered from cell x to gate nodes to pass data a,b and c to the graph. Thus, the expression is evaluated in applicative order by the by-value mechanism.

By-reference mechanism also naturally implements the graph sharing in reduction. For example, suppose the expression $x + x * x$, where $x = (a+b)*(b-c)$, is evaluated by demand driven (Fig.6). Even if operations + and * issue demands to the cell of variable x, the evaluation of x is triggered only once by the first arrived demand, and its result value is returned back directly to subsequent demands.

3.3 Nonstrict evaluations

Nonstrict function application is implemented using the by-reference mechanism. The function argument which is required to be evaluated lazy are not evaluated but its data flow graph, which is a recipe, is passed to the function body. In this case, a cell that points to the data flow graph (i.e., program code) is treated as a token. Consider the *ite* function as an example. The data flow graph of $ite(p, x, y)$ is represented as Fig.7.

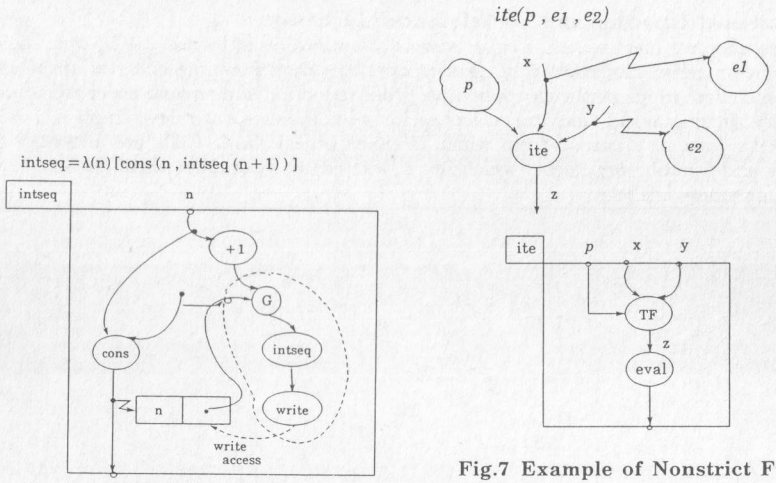


Fig.7 Example of Nonstrict Function

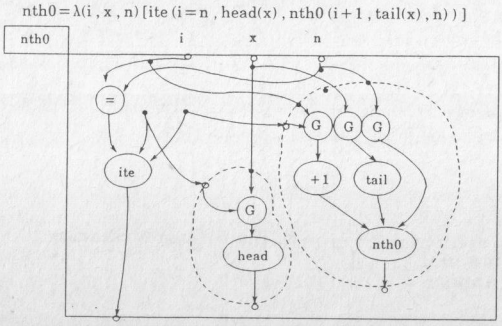


Fig.8 Data Flow Computing of Nonstrist Evaluation

In this scheme, arguments x and y have data flow graphs as their values, while p is an evaluated Boolean value. The Boolean p controls the inbound switch to select x or y, and put it to z. Eval operator evaluates the data flow graph pointed to by z. This mechanism implements the normal order reduction for *ite* function described in Section 2.1.

Nonstrict evaluation on structure data is also implemented on the data flow computation mechanism. In order to control the nonstrict operation of structure data, the read-ready tag and the demand-trigger tag are also used for data cells. In the nonstrict *cons* operation, its head and/or tail argument is not evaluated. Instead, the unevaluated data flow graph of the argument is written into head and/or tail field of the cons cell. The demand to evaluate the head and/or tail expression is issued when *head* and/or *tail* operation has accessed head and/or tail field of the cons cell [12,18,19]. As an example of the usage of nonstrict cons mechanism, a data flow implementation of the function *intseq* and *nth*0 is shown in Fig.8.

3.4 Supercombinator and Data Flow Graph

As described in Section 2.3, supercombinator is a lambda abstraction which has no free variables. The data flow model has quite similar properties to the supercombinator. All tokens in a data flow graph of function body are given explicitly as parameters, and no free variables exist in the data flow graph.

Data flow computing model, therefore, naturally implements supercombinator as a defined function. Eager and lazy evaluation mechanism in data flow computing model would enable the partial evaluation described in Section 2.3, as shown in Fig.9.

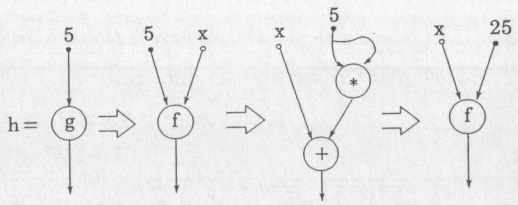


Fig.9 Partial Evaluation

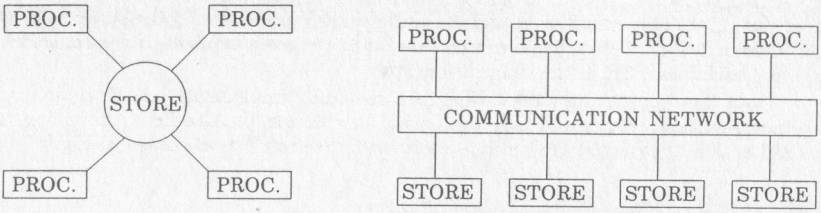
4. PARALLEL REDUCTION MACHINE

As opposed to the serial reduction control, stack based execution control mechanism does not work well for parallel reduction control. Instead, a packet based execution mechanism would be considered as a basis for parallel reduction control. The typical example of this type of implementation is ALICE machine[3]. ALICE machine is a physical implementation of graph reduction process described in Section 2. The ALICE machine structure is similar to that shown is in Fig.10(a).

The processors access the store, select a packet, perform a reduction and write new packet or modify old ones in the packet store. Each packet has three modes; active mode, suspended mode and dormant mode. The packet in active mode, indicated as !, is a candidate for further reduction. The packet in suspended mode, indicated as #n, is waiting for n of its arguments to reach where reduction is possible. The packet in dormant mode, indicated as -, is not currently a candidate for any action. An intelligent packet store could detect when packets had all the required arguments and make them processable. It could also detect when packets were no longer required and mark them for garbaging. In the physical implementation the packet store is partitioned and distributed within the machine as shown in Fig.10(b).

One of the problem in ALICE machine architecture is the overhead of operations in packet store. When a reduction is performed on the packet store the processing agent must rewrite the argument field. When a reduced value is returned back, the agent must check whether the suspended packet has changed to active packet, and if so, the agent must put the the active packet to an active packet queue. The key problem in implementation is how to reduce the overhead in these operations on packet store.

The Flagship machine [20] is a successor to ALICE machine. The Flagship is designed to improve the above mentioned overhead in ALICE machine. In the Flagship architecture, redundant demand propagation is assumed to be eliminated at compile time, and run time execution could be mainly controlled by data driven. However, mechanisms of access and operation on packet store are essentially same as those in ALICE machine, except that Flagship has a local packet store, instead of the global shared packet store, as shown in Fig.11. Therefore, Flagship still has the problem of overhead in data driven control and packet store operations.



(a) Abstract machine structure (b) A simple distributed machine structure

Fig.10 ALICE Machine Architecture

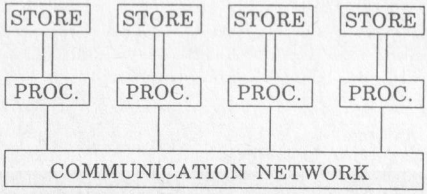


Fig.11 Flagship Machine Architecture

5. DATA FLOW BASED REDUCTION MACHINE

The important point in designing the parallel reduction machine is that the demand propagation in normal order reduction will be possibly eliminated in compile phase. By eliminating the redundant demand propagation, compiled code of a given functional program is executed essentially in data driven, except for a few part which needs to be executed in demand driven. Program analysis and compiler construction technique for optimizing the demand driven computation [14,15] is also an important issue, but those are omitted in this paper because the theme is beyond the scope of this paper.

In this section, an implementation model is proposed, which will reduce the overhead

in data flow control while preserving the parallelism maximally. The key idea is to extract a multi-thread control flow from the original data flow.

5.1 Features of Data Flow Architecture

Data flow architecture offers several beneficial features as the basis of parallel machine architecture [10,11,12].

(1) The data driven control mechanism is quite natural for parallel execution. It can maximally exploit the parallelism inherent in a given program.

(2) The tagged token mechanism, with which multi-process execution environments are managed on the hardware level, can efficiently control the parallel and concurrent execution of multiple function activations.

(3) The packet-communications-based circular pipeline architecture realizes pipeline and parallel operations between hardware modules, and this mechanism offers a basis for massively parallel MIMD machine architectures.

At the same time, the data flow architecture has several drawbacks:

(1) Communications overhead: There exists an overhead in communications that occurs within a processor and between processors. Low level communications are essential in data flow architecture, since all operations explicitly transfer data to their successors as tokens. (Here, it should be noted that the inter-processor communications overhead is not a problem in the data flow machine only, but is also essential in all multi-processor systems where communications are necessary among processors.)

(2) Overhead of fine grain data flow control: If the process granularity is set at a finer level, a larger number of switch and gate operations will be required. This results in an increase in data flow control overhead.

(3) Demerits of memory-less concept: The pure by-value data driven control mechanism produces needless data copies and redundant operations. A by-reference data sharing mechanism, on the other hand, never produces needless data, and therefore only the demanded operations are activated, since data are accessed only when they are needed. Furthermore, the data sharing and access-by-needs concept is essential for implementing the the lazy evaluation.

5.2 Implementation Model

In the practical implementation, the by-reference and by-value mechanisms are used for data tokens. All data tokens flowing in a data flow graph, unless having only one destination, denote memory cells that hold values, except for Boolean values and gate control signals. Boolean values and control signals are represented in one-bit data, and they are used directly for gate and switch controls [17].

This cell token flow implementation model offers the following good features compared with the conventional data flow model:

(1) Even if the result value produced by an operation is referred to by several operations, it is not necessary to create copies of it, since the by-reference mechanism permits those operations to share the operand data. The operand data is accessed only when it is needed.

(2) Any two nodes connected with a link have a data dependency relation. In the cell token flow model, the connection link between two nodes represents a memory cell address. The address of the memory cell can be determined at compile time, and thus be free of the run time memory allocation.

(3) The by-reference mechanism assures that an operand data of some operation exists when it is produced (and written into its memory cell) by the preceding operation. Therefore, the availability of an operand, call it x, need not be checked, if its partner operand is a dependent of operand x. This reduces the overhead of checking the pair token availability in detecting enabled instructions.

Considering these characteristics of the cell token flow model, we can eliminate redundant token flow links, and can transform the data flow program to a multi-thread control flow program, which preserves the original data dependency.

5.3 Control Flow Extraction

The multi-thread control flow program is extracted from a functional style source program in the following process. First, the source program is translated into a data flow graph. Then, the data flow graph is transformed to a multi-thread control flow program, through memory cell assignment and data flow optimization. (It should be noted that memory cell assignment and multi-thread control flow extraction can be performed on the source program directly. This is the issue of compiler construction technique, therefore detailed discussion is omitted here.) An example of the data flow program and its extracted multi-thread control flow program is shown Fig.12. An architecture of the processor which executes the extracted multi-thread control flow program will be described in later.

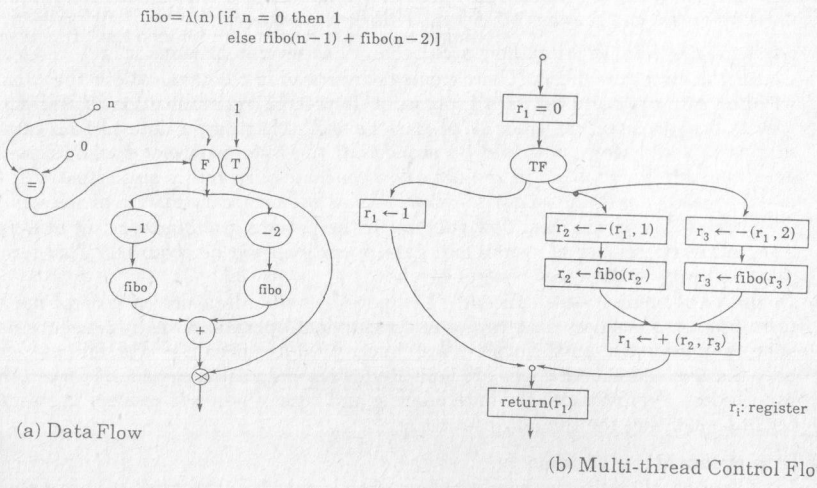


Fig.12 **Data Flow Program and its Extracted Multi-thread Control Flow Program**

5.4 Function Activations and Memory Blocks

In the course of computation, a new function instance is created for each function activation. These function instances share the same function body (program code). A memory block, which is used as a working register file, is assigned to an activated function instance. All operand data used in the instance are stored in the allocated memory block.

Each memory cell in a block is accessed by a local displacement address in the block. This local displacement is determined at compile time. At run time, memory cells are accessed with a two-level addressing mechanism; the dynamically specified block address and a statically decided local displacement.

5.5 Partial Evaluation and Function Closure

The evaluation environment for a newly created function instance is maintained in the memory block assigned to the function instance. In the course of execution, graphs always shrink except for function application, and when a function application occurs, a new memory block is allocated to the newly created function instance. Therefore, during execution, a larger memory block is never needed than was initially allocated.

5.6 Demand Driven Lazy Evaluation

Memory cells have two tags, r and c, for controlling the lazy evaluation discussed

in Section 3. Tag r shows whether the memory cell is a data value or code pointer. Tag c shows whether the memory cell has been demanded already or not. When an eval operator $eval(v)$ is activated, it accesses a memory cell allocated to variable v. If the cell has a value, i.e., r is on, the data flow graph has already been reduced to its value. If the cell has no value and has not been accessed yet, i.e., r and c are off, then a demand signal is delivered toward the entry point of data flow graph to start the evaluation. If the cell has been demanded already, i.e., r is off and c is on, the reduction has already been triggered by some other eval operator, and the eval operator waits for the value to be written to the cell.

6. MACHINE ARCHITECTURE

6.1 System Architecture: Parallel and Pipeline Multi-processor System

An architecture of parallel reduction machine is shown in Fig.13. The machine is constructed with a number of processing elements and structure memory elements. Structure memory is set as a local memory for a specific processor. At the same time, the structure memory can be accessed from other processors. The local/global access switching is simply controlled using the structure memory address.

Processors and memories are connected with multi-stage packet switching network. Processing elements, called PEs, execute a program. The program code and operand data are stored in Instruction Memory and Data Memory in the PEs. Structure data such as lists and arrays, on the other hand, are stored in structure memory, called SM, and the pointer to structure data stored in SM is treated as operand data within each PE. This mechanism for handling structure data is the same as that in the data flow machine DFM [18,19].

The inter-processor connection network is designed so as to realize a flexible tree structure in logical, while its physical connection is set regular and local. The network also controls the processor work load to balance among processor resources. The network watches the work load of each processor, and allocates an activated function instance to the least loaded processor so as to distribute the work load over processor resources.

6.2 Processor Architecture: Multi-thread Control Flow Executor

The design philosophy of the processor is as follows:

(a) Separation between data token and control signal: Only control tokens are used for execution control. Data values are stored in memory cells or pipeline latch register. The address of operand data is specified in the operand field of an instruction. Boolean values are treated as control token, so that switch operation is operated in the execution control module.

(b) Large capacity register file: The concept of a large capacity register file is used for implementing the operand memory system. The register file is divided into multiple blocks. Each block is assigned to each function instance at run time, while the local cell address in each block is obtained at compile time.

The processor is designed on the basis of a circular pipeline architecture constructed with Function unit (FU), Link memory unit (LM), Result write unit (RW), Instruction memory unit (IM), Activation control unit (AC), and Operand data memory (DM). These units are connected in a circular pipeline, as shown in Fig.14.

The execution of an instruction in the circular pipeline proceeds as following:

(1) The function unit executes operations specified by the operation code *opn* in Instruction packet (IP). The execution result is constructed as a Result Packet (RP), and sent to the RW.

(2) The RW writes the result data (*data*) to Operand data memory (DM). At the same time, the LM decides the IM-address of its destination operation node (*opn*), and constructs an Operand packet (OP).

(3) Instruction memory unit (IM) fetches the instruction code (*opc*), operand address (*opr*) and the destination node name (*des*) from the Instruction memory cell speci-

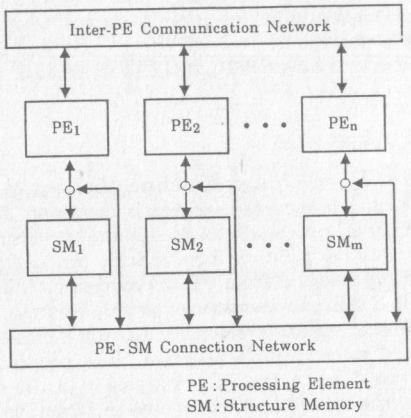


Fig.13 System Architecture

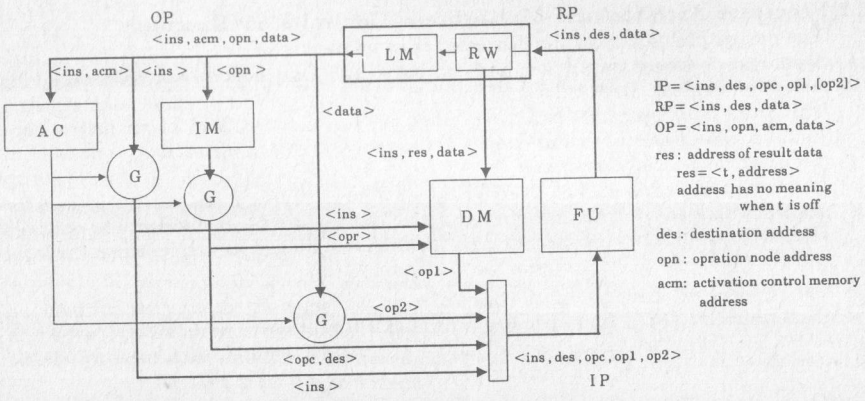


Fig.14 Processor Architecture

fied by the address *opn*.

(4) Activation control unit (AC) checks whether the partner operand has been written in Operand data memory. AC memory is used for token matching. AC memory is also divided into multiple blocks, each of which is assigned to a function instance. The AC memory cell is constructed with only two bits. One bit is used as a Boolean value, and the other bit is used as a tag to show the presence of a partner operand. The AC memory cell is accessed by two level address; the block address specified by the instance name *ins*, and the cell address *acm* which is local to the block. It should be noted that almost all two-operand operations need not examine the matching of pair tokens, as discussed in section 5.3. Only a few two-operand instructions are needed to examine the readiness of the pair operand.

(5) For enabled instructions, their operand data are fetched from DM. The DM cell is accessed by two levels of address: the DM block address which is specified by the instance name *ins*, and the local displacement in the DM block which is specified by the operand address *opr*. One problem is access contention to DM since DM access occurs at both result data write and operand data fetch. However this problem can be solved designing a multi-port memory device with high access speed using SRAM device technology.

7. CONCLUSIONS

A parallel graph reduction model and its machine implementation were discussed. First, a parallel graph reduction mechanism and its relation to data flow computing were discussed. In this paper, an extended data flow model, which introduces by-reference concept, was proposed. In the discussion, it was pointed out that the data flow computing model with by-reference concept was a natural implementation of parallel graph reduction. Then, an implementation of a parallel reduction machine was presented as a practical model. In the implementation, a cell token flow model was used. By using the cell token flow model, a given data flow program can be transformed to a multi-thread control flow program.

Last, an overview of the machine architecture was presented. The machine is designed on the basis of the packet communications concept, in which many processor and memory resources are connected with packet communications network. The processor is designed as a circular pipeline processor, which performs high speed execution of the multi-thread control flow program

Another important issue, which was omitted in this paper, is program analysis and compiling technique for functional program; the algorithms to analyze data dependency and nonstrictness, to optimize the functional program and to extract a multi-thread control flow.

By integrating researches on the parallel machine architecture discussed in this paper and compiler construction technique for functional language, a high performance parallel reduction machine will be developed as a future supercomputing system.

References

[1] Berkling,K.: Reduction language for reduction machines, Proc. IEEE Int. Symp. Comp. Arch., 1975, pp.133-140.

[2] Mago,G.: A cellular computer architecture for functional programming, Proc. IEEE Compcon, 1980, pp.179-187.

[3] Darlington,J. and Reeve,M.J.: "ALICE": A multiprocessor Reduction Machine, Proc. Conf. Functional Programming Languages and Computer Architecture, 1982, pp.65-75.

[4] Vegdahl,S.: A survey of proposed architecture for the execution of functional languages, IEEE Trans. on Computer, 33, 1984, pp.1050-1071.

[5] Watson,I.: Parallel data-driven graph reduction, Fifth Generation Computer Architecture, J.V.Woods, ed., North-Holland, 1985, pp.203-220.

[6] Ashcroft,E.A. and Jagannathan,R.: Operator nets, ibid. pp.177-201.
[7] Landin,P.J.: The mechanical evaluation of expressions, Computer Journal, 6,4, 1966, pp.308-320.
[8] Kieburts,R.B.: The G-machine: A fast, graph-reduction evaluator, Proc. Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201, 1985, pp.400-413.
[9] Turner,D.: A new implementation technique for applicative languages, Software-Practice and Experience, 9, 1979, pp.31-49.
[10] Dennis,J.B.: First version of data flow procedure language, Lecture Notes in Computer Science 19, 1974, pp.362-376.
[11] Arvind, Gostelow,K.P. and Plouffe,W.: An asynchronous programming language and computing machine, TR-114a, Dept. Information and Computer Science, Univ. California Irvine, 1978.
[12] Amamiya,M.: Dataflow computing and eager and lazy evaluations, New Generation Computing, 2, 1984, pp.105-129.
[13] Hughes,J: Super combinator - A new implementation method for applicative languages, Proc. 1982 ACM Symposium on Lisp and Functional Programming, pp.1-10, 1982.
[14] Ono,S., Takahashi,N. and Amamiya,M.: Optimized demand driven evaluation of functional programs on a dataflow machine, 1986 Int. Conf. on Parallel Processing, 1986.
[15] Clack,C. and Peyton Jones,S.L: Strictness analysis - A practical approach, Proc. Functional Languages and Computer Architecture, Lecture Notes in Computer Science 201, 1985, pp.35-49.
[16] Johnsson,T: Lambda lifting: Transforms programs to recursive equations, Proc. Functional Languages and Computer Architecture, Lecture Notes in Computer Science 201, 1985, pp.190-203.
[17] Amamiya,M: A new parallel graph reduction model and its machine architecture, Proc. France-Japan Artificial Intelligence and Computer Science Symposium 86, to be published by North-Holland, 1987.
[18] Amamiya,M., Hasegawa,R.,Nakamura,O. and Mikami,H.: A list-processing-oriented data flow machine architecture, Proc. NCC, AFIPS, 1982, pp.143-151.
[19] Amamiya,M., Takesue,M., Hasegawa,R. and Mikami,H.: Implementation and evaluation of a list-processing-oriented data flow machine, Proc. 13th Ann.Int. Conf. on Computer Architecture, 1986, pp.10-19.
[20] Watson,I, Sargeant,J, Watson,P and Woods,V.: Flagship computational models and machine architecture, ICL Technical Journal, pp.555-574, May, 1987.

PERFORMANCE ISSUES IN DATAFLOW MACHINES

John GURD, Wim BOHM *and* Yong Meng TEO

Department of Computer Science, University of Manchester
Oxford Road, Manchester M13 9PL, England

Issues affecting the performance of dataflow computers at the machine and language levels are explored. It is suggested that performance is dictated by the nature and the means of identification, distribution and control of workload in the hardware system. Dataflow is an asynchronous concurrent notation based on fine-grain message-passing in graphical programs. Dataflow machines comprise multiple processing elements and structure store modules connected together via a packet-based switching network. Workload is in the form of fine-grain data packets which trigger instruction-level activity in the various components of the hardware architecture. Workload is identified by a compiler for a high-level, single-assignment language, and is distributed across the hardware components dynamically at run-time. The amount of work at any instant can be controlled by a parallelism "throttle". The paper studies the performance of one example of a dataflow computer, the Manchester Dataflow Machine (MDFM).

1. DESIGN ISSUES FOR PARALLEL COMPUTERS

1.1. Architectural Levels

This paper explores issues affecting the performance of a particular style of parallel computer system. In considering such issues, it is important to distinguish between different *levels* of the system architecture.

Broadly speaking, computer systems are designed at three levels. The first, usually thought of as the lowest level, is that of the hardware architecture and its associated machine-code. This is often referred to as the *machine level*. The next, intermediate level is that of the so-called high-level programming language, from which machine-code is generated. This is often referred to as the *language level*. Most often, this is not at a sufficiently high level to express specific applications in a convenient way. Consequently, the highest level, known as the *applications level*, is constructed for particular application areas on top of the language level. The interface to this level is a kind of "very-high-level", applications-oriented language.

As far as the end-user of a computer system is concerned, performance at the applications level is most important. However, as this varies significantly according to the particular application, and in a somewhat unpredictable fashion, most attempts at evaluating performance are made at the two lower levels. This paper broadly conforms to this pattern, but an attempt is made to relate the performance assessment to a specific set of applications, the so-called *scientific problems*, which involve a substantial amount of numerical calculation.

1.2. Parallel Workload

Parallel system performance is predominantly determined by design decisions made at the machine and language levels. The most critical of these decisions concern the nature, identification, distribution and control of the parallel *workload*.

Different basic models of parallel computation can create workloads of a fundamentally different *nature*. For example, although many parallel programs are written as systems of communicating parallel *processes*, there is a critical distinction between systems that implement the communication using shared memory and those that use direct message-passing. The workload in the former involves accesses to a hardware third party (the shared memory) that are not necessary in the latter. Also, there is an important difference between such *process-based* schemes, which split the overall workload into *coarse-grain* chunks, and systems, such as dataflow, that segregate work into very *fine-grain* components. The former require infrequent transmission of usually large amounts of data, whereas the latter involve frequent communication of usually small amounts of data.

The *identification* of parallel workload is ultimately the responsibility of the compiler that translates a program from the language level to the machine level. However, certain systems, particularly among those based on dividing the overall problem into coarse-grain processes, require that the programmer be to some extent involved with this process, by *explicitly* identifying those parts of his program that can be executed concurrently. Perhaps a more sophisticated approach is to allow the compiler to derive parallelism *implicitly* from the program. However, conventional programming languages, with their in-built assumptions of sequential execution based on a globally addressable store, and the ability to address individual store locations using distinct program level names (the so-called problem of *aliasing*), often hinder the necessary analysis. Recent developments in language design have addressed this problem by abstracting high-level languages away from the machine level. In principle, the use of program transformation and optimisation techniques to create parallel versions of programs is feasible. In practice, this art is in its infancy, and few such systems are yet in general and reliable use. The identified parallel workload can be of two fundamentally different kinds. In the first, the amount of parallelism is completely determinable in advance of running the program. In the second, the amount of parallelism depends on complex run-time characteristics, and cannot be predicted without running the program.

The problem does not end with the identification of a parallel workload. The different parallel tasks still have to be *distributed* across the various parallel hardware components (processors and memories). Once again, the programmer may have to be involved in the process, explicitly deciding which parallel parts of his program should be executed in which part of the hardware. In making such decisions, the programmer will be concerned to balance the amount of work in distinct hardware components, and to minimise the amount of communication between components, insofar as the actual workload can be predicted. It is already clear that this is very difficult where the number of parallel processes exceeds a few tens and when the degree of parallelism is unpredictable prior to run-time. Consequently, systems are emerging that make such decisions independently of the programmer. Some such systems determine the distribution of work at compile-time, usually with results similar to those achieved by the programmer. Other systems distribute the workload at run-time, scheduling processes to processors at the time they are created, either *statically*, by a predefined algorithm, or *dynamically*, according to the then current activity level in each part of the hardware. Complex difficulties arise here, due to the need to communicate with other active processes and to minimise the amount of communication traffic in the hardware system.

In many circumstances it is possible for a very high degree of parallel activity to be identified in a program. Where the amount of this software parallelism exceeds the corresponding amount of parallelism available in the hardware, it is usually inefficient to allow all possible concurrent processes to execute. Consequently, there is a need to *control* the amount of active parallelism generated during the program run. This effect has been most noticeable to date in fine-grain parallel systems. However, it is clear that a similar phenomenon is possible in dynamic coarse-grain systems, such as processor "farms".

1.3. Absolute and Relative Performance

Traditionally, performance is rated by manufacturers at the machine level, using measurements such as MIPS (millions of instructions per second), usually given as absolute maximum values (guaranteed never to be exceeded). Such rates are *absolute*, in that they express direct hardware performance in machine level terms (instructions are not a recognisable concept at language level).

In order to get closer to the applications level, a measure related to the application area is required. For example, in scientific computations, absolute performance can be measured in MFLOPS (millions of floating-point operations per second). Similarly, symbolic processing rates are often measured in LIPS (logical inferences per second). Such measures do not necessarily correspond directly to hardware activity, since other operations are usually invoked to support these high level actions.

It is often useful to relate machine level activity to these higher level measures, for example by assessing the average number of instructions executed for every floating-point operation, or for every logical inference. Such *relative* measures give an idea of the effectiveness of a particular machine architecture, together with its compilation system, for a particular application area. They are useful because they discount the specific technology used to fabricate a machine.

2. DATAFLOW SYSTEM DESIGN

2.1. Introduction

Parallel dataflow systems have been the subject of detailed research since about 1970. Over the past decade, a widespread consensus has been achieved by the dataflow research community concerning the basic underlying model of computation, the organisation of machines, and appropriate high-level programming languages. The principal sites at which practical dataflow research is being conducted, and from which the following presentation is drawn, are the Massachussetts Institute for Technology (MIT) in the United States of America [4], the ElectroTechnical Laboratory (ETL) in Japan [20,24], and the University of Manchester in England [12,13]. The following sections express the consensus between these groups by inspecting an example of a practical dataflow computing system, the Manchester Dataflow Machine (MDFM), that has been implemented and evaluated at the authors' institution. Where consensus does not yet exist, the alternatives under consideration at different sites are presented. The presentation follows the pattern of section 1.2 by concentrating on the nature of the dataflow workload, and the means of its identification, distribution across and control within the parallel hardware resources.

2.2. Language Level

At language level, the consensus is to compile code from (usually simple) functional programming languages that are without the sequential and aliasing constraints of conventional languages. Standard functional languages, such as Lisp or ML, are not used because they allow manipulation of store locations in the conventional way. Hence, *pure* functional languages are preferred, but even the commonly used of these, such as Hope and Miranda, are not used by dataflow projects because they are centred on list-structured data, rather than arrays. The style of language most favoured by the dataflow research community is that of the *single-assignment languages*.

Single-assignment languages allow variables to be defined just once in a program, and, as a consequence, they can be read mathematically (as equations) just as easily as they can be read operationally. Examples of dataflow-oriented single-assignment languages are *Val* [1] and *Id* [2], both used at MIT, *DFC* [25], used at ETL, and *SISAL* [16], used at Manchester. Id is the most ambitious of these languages, featuring higher-order functions and complex loop structures. SISAL is a straightforward, orthogonally designed single-assignment language. DFC is still at the conceptual stage of design. It will use the syntax of C, but with single-assignment restrictions, although it is not yet clear how data structures fit into this framework.

SISAL has been designed jointly by researchers from Lawrence Livermore National Laboratory, Digital Equipment Corporation, Colorado State University and the University of Manchester. It was designed to act as a general-purpose non-sequential programming language for experimenting with parallel implementations of various scientific and numerical applications. A front-end compiler converts SISAL code into a graphical, machine-independent intermediate format, known as *IF1* [21]. Several standard optimisations, such as eliminating common sub-expressions and moving constants out of loops, are performed by IF1-to-IF1 transformations [22]. Machine-specific code generators convert the IF1 code into appropriate machine-code for a variety of parallel machines, including the Cray-1, the Sequent Balance, and the MDFM.

An example of a compact SISAL program is the matrix multiplication program, shown in figure 1, which is used to illustrate this paper. Note that Matrix B is in transposed form.

2.3. Model of Computation

There are several versions of the dataflow model of computation [3,9,11], but the consensus is to use the *tagged-token* variant. In all dataflow models, machine-code programs are represented as directed graphs in which nodes represent instructions and arcs represent data dependencies. Data is transported along arcs by special vehicles, known as *tokens*, each one of which comes into existence solely for the purpose of carrying one data value from the node that creates it to a subsequent node that needs it. Where a data value is needed by more than one subsequent node, multiple tokens, each carrying the same value, are sent along multiple arcs to the appropriate nodes. In the tagged-token model, each token also carries a *tag* that distinguishes it from every other token on that arc. Tags are generated systematically by specialised instructions so that graphical code can be re-used for cyclic and recursive programs [11]. Tags are also used to distinguish elements of data structures. In the MDFM, a tag has two fields, known as the *activation name* (AN - used to separate function calls or loop bodies) and the *index* (IX - used to separate elements of data structures). Nodes are prohibited from generating more than one token per arc with the same tag.

A node is permitted to execute only when a *matching* set of tokens is present on its input arc(s). This criterion is fulfilled when every input arc contains a token carrying the same tag (additional tokens with different tags are ignored). The matching set of tokens is collectively removed from the input arc(s), and the node is executed in such a way that it eventually produces appropriate result tokens on its output arc(s). Conditionals (and cycles) are handled by *branch* nodes with multiple output arcs that receive result tokens selectively, according to some computed condition.

```
export Main

type Vector = array[real]
type Matrix = array[Vector]

function matmult(A,B : Matrix returns Vector)
for Row in A
  AnsRow :=
    for Col in B
      El :=
        for Rel in Row dot Cel in Col
          R := Rel * Cel
        returns value of sum R
        end for
    returns array of El
    end for
returns value of catenate AnsRow
end for
end function

function Main(N: Integer returns array[real])
let
   MRow := array_fill(1,N,1.0);
   M    := array_fill(1,N,MRow);
   PRow := array_fill(1,N,2.0);
   P    := array_fill(1,N,PRow)
in Matmult(M,P)
end let
end function
```

FIGURE 1
Matrix multiplication program in SISAL

Finally, the consensus is that data structures should be stored in a *data structure store*, and represented by *pointer tokens* when processed in their entirety. Where they need to be processed element-by-element, they can be accessed in the data structure store, either individually or in vector style, using specific *read* and *write* instructions [5,18,19,20].

Figure 2 shows the graphical form of a piece of MDFM machine-code that is typical of tagged-token dataflow machines. It shows optimised code generated for the inner loop of the SISAL matrix multiplication program of figure 1. The code in figure 2 is a detailed version of that given by Böhm and Sargeant in figure 16 of [7]. Each instruction is prefixed by a letter indicating the nature of its output. "N" shows that the instruction has only one logical output value, although this value may be used twice (as in the case of the DUPlicate instruction, for example). Other instructions have two logical output values, of which either the left ("L"), the right ("R") or both ("D") may be selected. The instructions used are as follows:

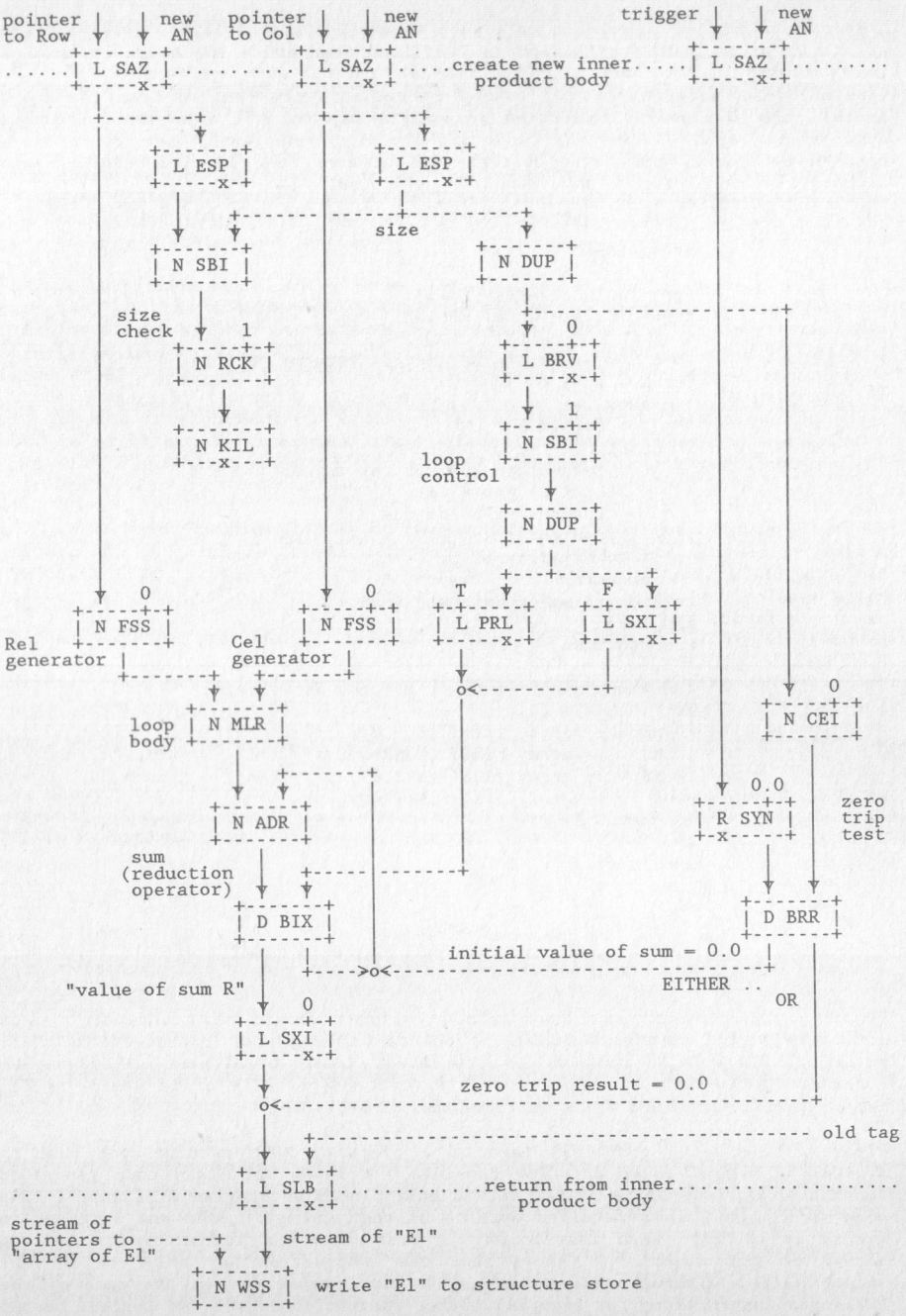


FIGURE 2

MDFM machine-code for inner product of matrix multiplication

DUP - DUPlicate incoming token (used to make multiple copies of tokens)
SYN - SYNchronise two tokens together (used to generate a constant token)
ADR - ADd together two Real-valued tokens
MLR - MuLtiply together two Real-valued tokens
CEI - Compare (Equal) two Integer-valued tokens (yields a boolean token)
SBI - SuBtract one Integer-valued token from another
BRR - BRanch (Repeat-type) according to value of r.h. boolean token
BIX - Branch according to value of r.h. boolean token and Increment indeX
BRV - BRanch according to zero/nonzero Value of r.h. integer token
ESP - Extract the Size of a structure from a Pointer token
FSS - Fetch Stream of tokens from Structure store
WSS - Write token to Structure Store
SXI - Set indeX using r.h. Integer token
SAZ - Set Activation name and Zero index
SLB - Set LaBel (i.e. change tag)
PRL - PRoLiferate l.h. token as a stream of tokens of length r.h. value
RCK - Range CheckK (used to test for incompatible matrix sizes)
KIL - KILl (i.e. destroy) all incoming tokens

These and the other instructions in the MDFM instruction-set are described in detail in the Basic Programming Manual for the MDFM [15].

2.4. Machine Architecture

The separation of data structure storage from instruction processing leads to machines with multiple *processing elements* (PEs) and *structure store modules* (SSs), connected via a packet-based *switching network*. In the case of the MDFM, a *global allocator module* (GA), responsible for resource management, is also included, and the whole system is attached to a conventional VAX 11/780 *host computer* (HC). In the case of the MIT machine, structure storage is contained within the PEs, and certain of the PEs are reserved for resource management.

The basic structure of the MDFM is illustrated in figure 3. In the PEs, program graphs are represented as lists of the instructions to be performed (the nodes) plus the addresses of their successor nodes (equivalent to the arcs). Each entry in the list corresponds to a conventional computer instruction. For example, figure 4 shows the literal MDFM machine-code corresponding to the graphical form shown in figure 2. This code is held in the *instruction store unit*. Each address in the store contains an instruction-code plus, where appropriate, one or two successor-instruction-address(es) (defining the output arc(s)) and/or a literal operand. Each successor-instruction-address specifies a *matching function* that is used when the corresponding output token reaches the appropriate token matching unit [12,15].

Tokens-on-arcs are represented by *token-packets* containing the data, the tag, and a successor-instruction-address. The process of grouping together a set of matching input tokens is performed in the *token matching unit*. The tag-plus-successor-instruction-address of each incoming token is compared against a set of *unmatched tokens*. Where a complete set of matching tokens is found, the set is sent to the instruction store unit to fetch the appropriate instruction.

Once this successor-instruction has been attached to its set of matching input tokens, the entire *executable packet* is sent to the *processing unit* to be executed. The data and tag(s) resulting from execution are attached to the new successor-instruction-address(es) to form the output token-packet(s). Output token-packets are sent to the token matching unit in a successor PE through the switching network and a *token queue unit*. The latter is provided simply as a buffer to hold excess token-packets when there is too much parallel activity.

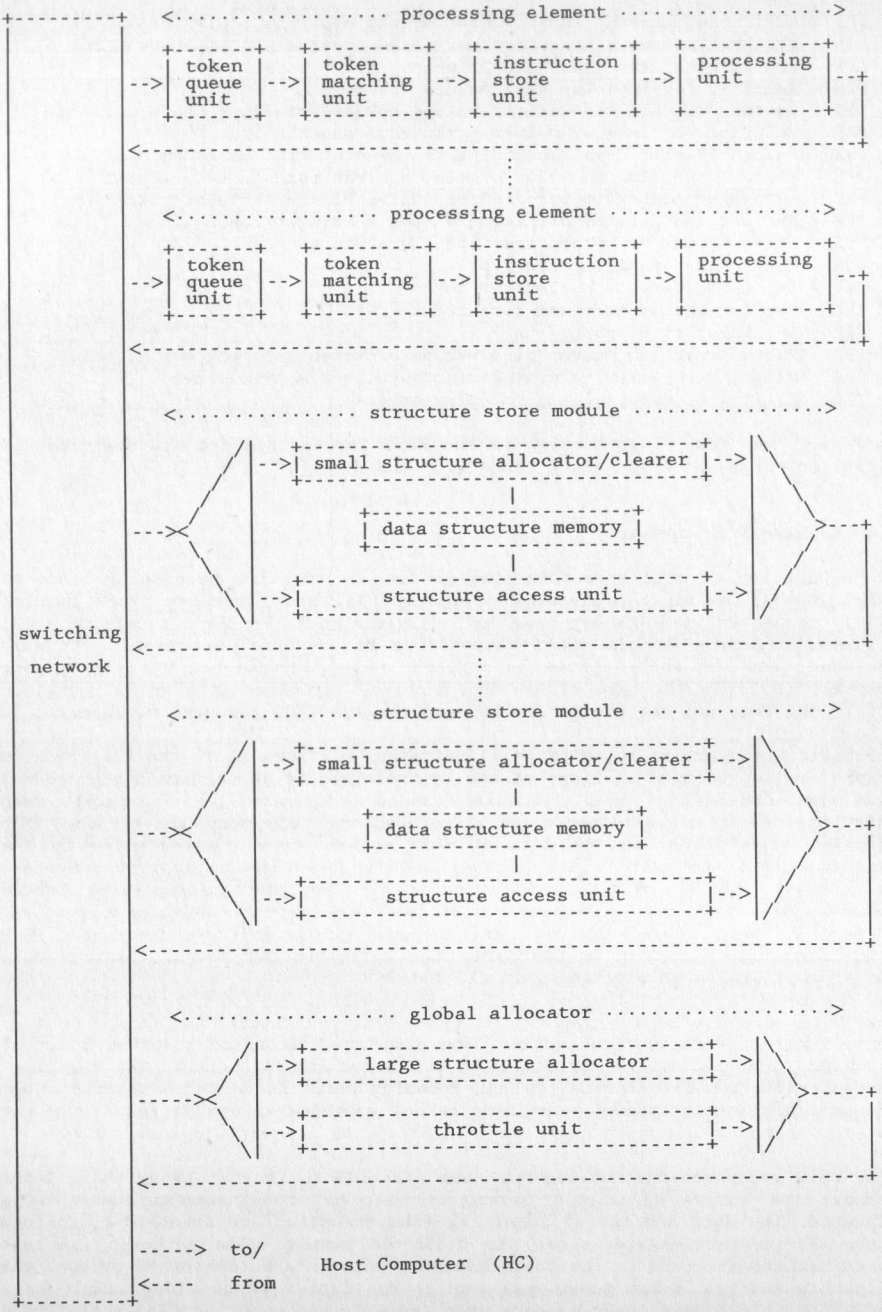


FIGURE 3

Multi-ring MDFM machine architecture

address	instruction code	destination address(es)	operand (type)	(value)
1 164:	L SAZ	1 190LBY	B	T
1 174:	L SAZ	1 181LBY 1 182LBY		
1 175:	L SAZ	1 183LBY 1 185LBY		
1 176:	L BRV	1 177LBY	I	0
1 177:	N SBI	1 178LBY	I	1
1 178:	N DUP	1 179RBY 1 180RBY		
1 179:	L PRL	1 194REW	B	T
1 180:	L SXI	1 194REW	B	F
1 181:	L ESP	1 186LEW		
1 182:	N FSS	1 189LEW	I	0
1 183:	L ESP	1 186REW 1 184LBY		
1 184:	N DUP	1 176LBY 1 191LBY		
1 185:	N FSS	1 189REW	I	0
1 186:	N SBI	1 187LBY		
1 187:	N RCK	1 188LBY	I	1
1 188:	N KIL			
1 189:	N MLR	1 193REW		
1 190:	R SYN	1 192LEW	R	0.0
1 191:	N CEI	1 192REW	I	0
1 192:	D BRR	1 193LEW 1 196LEW		
1 193:	N ADR	1 194LEW		
1 194:	D BIX	1 195LBY 1 193LEW		
1 195:	L SXI	1 196LEW	I	0
1 196:	L SLB	1 201LEW 1 216REW		
1 216:	N WSS			

FIGURE 4
Literal MDFM machine-code corresponding to the graph of figure 2

A structure store module (SS) responds to four kinds of token-packet that are normally generated by instructions executed in the PEs. The first kind of packet is a *structure request*, which causes the SS to allocate space for a data structure. Request packets for *large* structures are sent to the global allocator, rather than an individual SS, so that the structure can be stored in *interleaved* fashion across all the structure memory units, to improve access speed [18,19]. In either case, a *pointer token*, representing the structure as a base address and an upper bound, is returned. The second kind of packet is a *write request*, containing a data value together with the address at which it should be written. The address is constructed by the PE using the original pointer token and the required offset within the structure. If the offset exceeds the upper bound, an error is signalled. The third kind of packet is a *read request*, which accesses a previously written element in a similar manner. If a read request arrives at a SS before the corresponding write request, the read request is automatically *deferred* and then processed later, after the appropriate write packet has arrived. The final kind of packet is a *reference count request*, which modifies the reference count associated with each data structure. These packets are issued by the PEs in line with the usual scheme for reference counting garbage collection [18,19].

Apart from handling requests for allocation of large, interleaved data structures, the global allocator is also responsible for issuing unique activation names (for use in tags) and for "throttling" the amount of parallel activity in the machine. The latter is achieved by controlling the release of new activation names which are requested by *process request* tokens sent from PEs executing GAN (Generate Activation Name) instructions [17].

There is as yet little consensus about the topology that is most suitable for the switching network that interconnects the hardware components. The Manchester team is experimenting with an equal-path-length switch that involves an equal delay for every packet transferred through it. The ETL team is using an hierarchical switch in which local transfers are delayed less than non-local ones [20,24]. The latter should be more efficient, provided that the language level compiler can plant code that exhibits suitable *locality*.

2.5. Workload Identification

The overall nature of dataflow workload is determined by the computational model and the basic machine architecture, which were outlined in sections 2.3 and 2.4, respectively. It consists of fine-grain (instruction level) activities in (a) the PEs, triggered by the matching together of data tokens in the token matching unit, (b) the SSs, triggered by structure handling messages generated in the PEs, and (c) the global allocator, also triggered by messages from the PEs. Tokens and other messages are transmitted around the hardware system by data packets.

In the MDFM, the workload is identified by the IF1 code generator. The total amount of processing and message traffic in each part of the system depends on the nature of the program and the value of data presented to it. Multiple activations of instructions are caused by (a) iterative cycling around SISAL *for initial* loops, (b) recursive calls to SISAL *functions*, and (c) execution of MDFM *proliferate-type* instructions, such as FSS (Fetch Stream from Structure store), planted to implement SISAL *for all* constructs. In the first two cases, tokens are distinguished by the AN field of their tags. In the latter case, the IX field is used. Apart from this, (tokens destined for) different instructions are distinguished by the address of the instruction in the instruction store. Examples of total workload in the MDFM are given in section 3.2.

2.6. Workload Distribution

Workload is distributed across the components of the hardware system dynamically at run-time, but different schemes are used in different machines. In the MIT machine and the SIGMA-1, some attempt is made to "cluster" instructions together in hardware modules, according to rules that maintain *locality* of communication [4,20]. This implies that tokens and instructions belonging to a logical *process* should be located in "adjacent" hardware modules. The hierarchical nature of the switching network in these machines defines the nature of adjacency that must be exploited by the workload distribution algorithm.

In the MDFM, the switching network is of uniform delay, and no advantage accrues when instructions are clustered. Consequently, a much simpler distribution algorithm can be used. A randomising *hash function* is applied to the tag-plus-successor-instruction-address of every token arriving at the switching network. The resulting hash address determines which of the PEs the token should be sent to. Small structure request messages can be treated similarly, with the hash address determining which of the SSs to use. Structure access messages are directed to an SS according to the way the corresponding structure has been allocated space.

The above scheme has the disadvantage that, where the tag is used to compute the hash function, code must be replicated in several of the PEs. However, it appears that the effectiveness of workload distribution is greatly improved by using the tag. Results showing this are presented in section 3.3. Corresponding results for the SIGMA-1 system, with its hierarchical switching network, are presented in [14].

2.7. Control of Parallelism

All dataflow researchers agree that some form of parallelism control is necessary, but the study of specific mechanisms for this is in its infancy, so there is little scope for consensus. The MIT group has published a scheme for controlling loop parallelism, using a method called *k-bounding* [6]. They also have a general resource management scheme whose precise details are not yet published. The Manchester group is studying a general parallelism control scheme which they call *throttling* [17]. The ETL group have yet to publish anything in this area.

Although it is not yet implemented in hardware, an overall structure for the MDFM throttle unit has been determined and extensively simulated [23]. The scheme relies on the controlled release of tokens belonging to an instance of a loop body or function call according to the level of activity in the machine when the instance is invoked. This is possible because the code planted for invoking such an instance involves a special instruction (GAN) to generate the new activation name (AN) for tagging tokens belonging to the instance. Instead of simply releasing the new AN on request, the system can choose to hold up this instance, by denying it an immediate new AN and releasing one later, when contention for resources has diminished. This is achieved by executing GAN instructions in the global allocator, which receives messages from elsewhere in the system to keep it informed of the overall level of machine activity. By allowing progress through the computation in a systematic fashion, the maximum amount of storage required can be significantly reduced. Some results demonstrating this are given in section 3.4.

3. DATAFLOW MACHINE PERFORMANCE

3.1. Benchmark Programs

A number of SISAL programs have been written and their performance on the prototype MDFM has been analysed [10]. A small number of these programs have been analysed in greater detail, using a multi-ring MDFM simulator [23]. The following subsections summarise the simulation results. The programs are as follows:

MM - The matrix multiplication program of figure 1. This features heavy use of 2-d arrays stored in the SS. The program is written to take full advantage of the vectorising optimisations in the SISAL code generator [7]. Input parameter *n* defines the size of the (square) matrices. The program generates its own input matrices before multiplying them together.

LR - Solves a simple version of Laplace's equation, over a square grid with fixed boundary conditions, using a straightforward relaxation method. Input parameter *n* defines the size of the square grid. The program generates its own initial data before the relaxation is repeated for 5 cycles. (46 lines of SISAL.)

ES - Solves a 2-d hydrodynamics problem, including a heat conduction phase, over an *n*-by-*n* square grid. This is by far the most complex of the test programs. It is a version of the SIMPLE benchmark, originally written in Fortran by staff at Lawrence Livermore National Laboratory [8]. The SISAL code was produced by Jim McGraw and Steve Skedzielewski during a visit to Manchester in 1983. This version is modified slightly to take advantage of a MDFM machine-code optimisation for accessing arrays whose lower bound is unity [7]. A single iteration is simulated. (1510 lines of SISAL.)

BI - Computes the integral of $(x^2 - 6x - 10)$ using a simple repeated binary split with trapezoidal approximation. Input parameter *n* defines the depth of splitting before the approximation is applied. Features "regular" divide-and-conquer recursion which requires good control of parallelism. (32 lines of SISAL.)

NQ - Solves the "n-queens" chess problem (i.e. works out how to place *n* queens on an *n*-by-*n* chess board in such a way that none of the queens checks one another). Features "irregular" divide-and-conquer recursion. (79 lines of SISAL.)

3.2. Total Workload

The total workload for the five benchmark programs is summarised in figure 5. The principal workload is expressed in terms of the total number of tokens passing through the switching network. The next eight columns show how that workload is distributed between the different kinds of hardware module. Only one of each type of hardware module is simulated, so that overheads are minimal. The column labelled S1 gives the total number of instructions executed, and the next column indicates the average number of tokens passing through the switching network for every executed instruction. The final two columns, labelled F1 and MMR, show the total number of floating point operations and the ratio of instructions executed per floating point operation [12], respectively.

program/ datasize	total tokens	% tokens received PEs	SSs	GA	HC	% tokens transmitted PEs	SSs	GA	HC	S1	tokens /instr.	F1	MMR
MM n=10	14272	81.1	16.9	1.2	0.7	83.3	15.8	0.9	0.0	7402	1.9	2000	3.7
MM n=20	86546	78.1	20.1	0.9	0.5	79.9	19.5	0.5	0.0	39822	2.2	16000	2.5
MM n=30	265422	77.3	21.5	0.8	0.3	78.6	21.1	0.4	0.0	115442	2.3	54000	2.1
LR n=15	52729	80.2	18.2	1.2	0.4	86.2	13.4	0.4	0.0	29249	1.8	5070	9.7
LR n=20	86226	78.9	19.6	1.1	0.5	85.1	14.6	0.3	0.0	46359	1.9	9720	8.0
LR n=25	127597	77.9	20.6	1.1	0.5	84.4	15.3	0.3	0.0	67069	1.9	15870	7.7
ES n= 5	70608	86.5	12.7	0.6	0.2	89.1	10.8	0.1	0.0	41396	1.7	5771	7.2
ES n=10	341654	85.5	13.6	0.6	0.2	87.5	12.4	0.1	0.0	191101	1.8	34853	5.5
BI n= 6	3312	98.8	0.0	1.1	0.0	99.3	0.0	0.6	0.1	2221	1.5	957	2.3
BI n= 8	13348	98.8	0.0	1.2	0.0	99.3	0.0	0.6	0.0	8941	1.5	3837	2.3
BI n=10	53488	98.9	0.0	1.1	0.0	99.4	0.0	0.6	0.0	35821	1.5	15357	2.3
NQ n= 5	68344	96.5	2.5	1.0	0.0	97.7	1.7	0.6	0.0	47922	1.4	-	-
NQ n= 6	288455	97.2	1.8	1.0	0.0	98.0	1.4	0.6	0.0	203359	1.4	-	-
NQ n= 7	1276914	97.1	2.0	0.9	0.0	97.9	1.6	0.5	0.0	897388	1.4	-	-

FIGURE 5
Total workload for five SISAL programs

The values in figure 5 are obtained by a fully timed simulation of the prototype MDFM hardware which contains a single PE and a single SS. The most essential control option (namely throttling [17]) is included. This gives the best performance on the MDFM with respect to store use [23].

3.3. Balance between Processing and Structure Accessing

Knowledge of the percentage workloads at the input and output of each kind of hardware module makes it possible to anticipate the worst-case average loadings and, hence, to achieve a balance between the amounts of hardware dedicated to each function. Clearly, the optimum balance also depends on the relative processing speeds of the various hardware modules. With timing figures obtained from the hardware prototype, the relevant simulation results are summarised in figure 6. The processing rates for each type of hardware module are expressed in millions of input tokens processed per second. The maximum PE processing rates are established by setting all timing delays for the other types of module to zero.

program/ datasize	PEtokens/ SStokens	program parallelism	PE processing rates max.	actual	eff.	SS processing rates actual
MM n= 5	6.3	16.7	2.11	1.71	81%	0.27
MM n=10	4.8	47.4	2.27	2.19	96%	0.46
MM n=30	3.6	117.0	2.31	2.29	99%	0.64
LR n=25	3.8	151.1	2.02	1.76	87%	0.47
ES n=10	6.3	179.9	2.36	2.23	94%	0.36
BI n=10	-	124.8	2.46	2.46	100%	-
NQ n= 7	48.4	159.7	2.36	2.36	100%	0.05

FIGURE 6
Processing and structure accessing rates for five SISAL programs

Several versions of program MM are shown first, to illustrate the effect of program parallelism [12]. As the program parallelism increases, the maximum PE processing rate approaches an asymptote determined by the PE pipeline structure and the nature of the MM program. For the other programs, the datasize is selected so that the corresponding asymptote has been reached. As the SS is not pipelined, program parallelism does not affect the SS processing rates in the same way. However, the proportion of different kinds of SS access changes (reads becoming more predominant as n increases) and this affects the maximum possible processing rate due to the large difference in processing time between reads and the other SS functions.

In most of the programs it is clear that the PEs are working at full rate while the SSs are under-utilised. For the LR and ES programs, structure store accesses are beginning to limit the PE processing rate. This indicates that there should be at least one SS per PE for these programs (or that a faster SS should be constructed). For the other programs it may be possible for one (slow) SS to support more than one PE.

3.4. Effectiveness of Workload Distribution

The effectiveness of the hashing method for distributing workload across multiple PEs and SSs is illustrated in figure 7. The number of SSs is always equal to the number of PEs. For the ES program, there are 4 PEs and 4 SSs. For the other programs, there are 16 PEs and 16 SSs. The processing rates are quoted per PE. The maximum possible rates are the actual processing rates given in figure 6. The efficiency (eff.) therefore measures the effectiveness of the workload distribution method. The programs marked * use a hash function computed using the successor-instruction-address only.

program/ datasize	tokens total	%GA	program parallelism	PE processing rates max.	actual	eff.	SS processing rates actual
BI n= 8	13280	0.6	101.6	2.46	1.54	63%	-
BI n=10	53318	0.8	317.0	2.46	2.17	88%	-
BI n=12	213254	0.8	673.0	2.46	2.43	99%	-
BI n=12*	213318	0.8	121.1	2.46	1.64	67%	-
MM n=35	410590	0.8	498.8	2.31	1.81	78%	0.52
MM n=35*	409780	0.6	43.9	2.31	0.55	29%	0.16
LR n=40	309601	0.5	290.9	2.02	0.98	49%	0.31
ES n=10	341400	0.5	201.7	2.36	2.13	90%	0.34
NQ n= 7	1273984	0.6	390.9	2.36	2.30	98%	0.05

FIGURE 7
Processing and structure accessing rates in a multi-ring MDFM system

Several versions of the BI program are shown first, to illustrate the effect of program parallelism. As parallelism increases (with *n*), the efficiency of PE usage approaches an asymptote determined by the workload distribution. For the other programs, the datasize is chosen so that the corresponding asymptote has been reached. The small increase in total token traffic and the small overall traffic to the GA indicate the low cost of distributing and throttling the parallel workload. The deleterious effect of not using the tag when computing the distribution hash address is apparent.

3.5. Effectiveness of Parallelism Control

The above experiments have all been conducted using the best available control of parallelism [17,23]. The results given in figure 8 show how necessary this is. Here, some of the programs from figure 7 are run with and without the throttle option. The maximum occupancies of the token stores in the token queue (TQ) units and token matching (MS) units in the PEs are shown. Unnecessarily large stores are required unless the throttle is used, except for the ES program, which has an unusual coarse-grain structure and is difficult to throttle in its current form.

program/ datasize	unthrottled TQ	+ MS	= total	throttled TQ	+ MS	= total
MM n=35	91911	+ 87122	= 179033	6764	+ 39500	= 46264
LR n=40	8185	+ 19793	= 27978	2225	+ 15015	= 17240
ES n=10	7665	+ 27282	= 34947	2564	+ 26803	= 29367
BI n=12	7781	+ 20941	= 28722	2819	+ 3883	= 6702
NQ n= 7	24545	+ 23446	= 47991	3593	+ 9689	= 13282

FIGURE 8
Store usage within the PEs with and without the parallelism throttle

As explained in section 2.7, the MDFM throttle works in a coarse-grain fashion, by holding back new ANs for functions and loop bodies when the level of machine activity is too high. As machine activity increases and diminishes, it is important that the throttle continues to release pending GAN requests at a sensible rate. Hence, a *throttle delay time* (i.e. the minimum possible time between the release of two successive new ANs) is calculated periodically by the throttle and used to limit the rate of release of new ANs. The optimum value of throttle delay time is governed by the time taken by the machine to process all the tokens "belonging to" a new AN. If ANs are released too quickly, the token stores in the PEs will fill up with excess tokens that will not be processed for a long time. If ANs are released too slowly, the token stores in the PEs will be starved, creating periods of inefficient inactivity. In the experiments cited above, the throttle delay time depends on the instantaneous lengths of the PE token queues and the total number of PEs.

4. CONCLUSIONS

We have presented dataflow in terms of the nature of its fine-grain parallel workload, and the means of identification, distribution and control of this workload. By conducting experiments with several benchmark programs, written in the single-assignment language SISAL and run on simulated versions of the Manchester Dataflow Machine (MDFM), we have demonstrated the effectiveness of the dataflow scheme. The total workload, and its distribution across the two major functions of processing and structure accessing, has been assessed. The question of balance between different hardware modules has been addressed. For modestly parallel systems, it has been shown that the workload can be automatically and effectively distributed across multiple hardware modules. We have also demonstrated the feasibility of controlling the proliferation of parallel work in such a system.

These performance characteristics are hard to achieve in a parallel machine design, and, to our knowledge, no other type of system has demonstrated them so convincingly. However, it must be stressed that these ideas are still at an early stage of development and there is much further work to be done. Although the throttle works well for most of the benchmark programs, it behaves badly for the ES program. The irregular size of the processes in this program makes it difficult for the throttle to settle into a steady rhythm. Consequently, large and small processes are released at the same rate, and the workload is not controlled satisfactorily. Also, the paper has failed to consider performance at the applications level, which is surely the most important of all. In particular, there is scope for restructuring the applications themselves in order to exploit hardware parallelism.

ACKNOWLEDGEMENTS

The authors are indebted to the members of the Dataflow Research Group at the University of Manchester. Dataflow research at Manchester has been supported by the Science and Engineering Research Council of Great Britain, the UK Alvey Programme in advanced information technology, and Digital Equipment Corporation. The SISAL language was designed in collaboration with Lawrence Livermore National Laboratory, Digital Equipment Corporation, Colorado State University and the University of East Anglia.

REFERENCES

[1] Ackerman, W.B. and Dennis, J.B., VAL : A Value-Oriented Algorithmic Language - Reference Manual, *Technical Report LCS-TR-218* (MIT, 1979).

[2] Arvind, Gostelow, K.P. and Plouffe, W., An Asynchronous Programming

Language and Computing Machine, *Technical Report ICS-TR-114a* (UC, Irvine, 1978).

[3] Arvind and Gostelow, K.P., Some Relationships between Asynchronous Interpreters of a Dataflow Language, *in*: Neuhold, E.J., (ed.), *Formal Description of Programming Concepts* (North Holland, Amsterdam, 1978) pp. 95-119.

[4] Arvind and Kathail, V., A Multiple Processor Dataflow Machine that Supports Generalised Procedures, *Proceedings 8th Annual Symposium on Computer Architecture* (ACM, 1981) pp. 291-302.

[5] Arvind and Thomas, R.E., I-Structures - An Efficient Data Type for Functional Languages, *Technical Memo LCS-TM-178* (MIT, 1981).

[6] Arvind and Culler, D.E., Managing Resources in a Parallel Machine, *in*: Woods, J.V., (ed.), *Fifth Generation Computer Architectures* (North Holland, Amsterdam, 1986) pp. 103-121.

[7] Böhm, A.P.W. and Sargeant, J., Efficient Dataflow Code Generation for SISAL, *Technical Report UMCS-85-10-2* (University of Manchester, 1985).

[8] Crowley, W.P., Hendrickson, C.P. and Rudy, T.E., The Simple Code, *Technical Report UCID-17715* (Lawrence Livermore National Laboratory, 1978).

[9] Dennis, J.B., First Version of a Data Flow Procedure Language, *Lecture Notes in Computer Science* 19 (1974) pp. 362-376.

[10] Foley, J.F., Benchmark Test Evaluation Report, *Technical Report UMCS-87-11-2*, (University of Manchester, 1987).

[11] Gurd, J.R., Fundamentals of Dataflow, *in*: Chambers, F.B., Duce, D.A. and Jones G.P., (eds.), *Distributed Computing* (Academic Press, London, 1984) pp. 3-19.

[12] Gurd, J.R., Kirkham, C.C. and Watson, I., The Manchester Prototype Dataflow Computer, *Communications of the ACM* 28/1 (1985) pp. 34-52.

[13] Gurd, J.R., Kirkham, C.C. and Böhm, A.P.W., The Manchester Dataflow Computing System, *in*: Dongarra, J.J., (ed), *Experimental Parallel Computing Architectures*, (North Holland, Amsterdam, 1987) pp. 177-219.

[14] Hiraki, K., Sekiguchi, S. and Shimada, T., System Architecture of a Dataflow Computer, *Proceedings 1987 Region 10 Conference on Computers and Communication: toward 2000* (IEEE, 1987) pp. 1044-1049.

[15] Kirkham, C.C., The Manchester Dataflow System - Basic Programming Manual, *Internal Report UMCS-DF-BPM* 6th edition (University of Manchester, 1987).

[16] McGraw, J.R. *et al.*, SISAL - Streams and Iteration in a Single-Assignment Language, *Language Reference Manual* (Lawrence Livermore National Laboratory, 1983).

[17] Ruggiero, C.A. and Sargeant, J., Control of Parallelism in the Manchester Dataflow Computer, *Lecture Notes in Computer Science* 274 (1987) pp. 1-15.

[18] Sargeant, J., Efficient Stored Data Structures for Dataflow Computing, *Ph.D. Thesis UMCS-85-8-2* (University of Manchester, 1985).

[19] Sargeant, J. and Kirkham, C.C., Stored Data Structures on the Manchester Dataflow Machine, *Proceedings 13th Annual Symposium on Computer Architecture* (ACM, 1986) pp. 235-242.

[20] Shimada, T. *et al.*, Evaluation of a Prototype Data Flow Processor of the SIGMA-1 for Scientific Computations, *Proceedings 13th Annual Symposium on Computer Architecture* (ACM, 1986) pp. 226-234.

[21] Skedzielewski, S. and Glauert, J.R.W., IF1 - An Intermediate Form for Applicative Languages, *Reference Manual M-170* (Lawrence Livermore National Laboratory, 1985).

[22] Skedzielewski, S. and Welcome, M., Data Flow Graph Optimisation in IF1, *Lecture Notes in Computer Science* 201 (1986) pp. 17-34.

[23] Teo, Y.M., Performance Evaluation of a Heterogeneous Multi-ring Dataflow Machine, *M.Sc. Thesis* (University of Manchester, 1987).

[24] Yuba, T. *et al.*, SIGMA-1: A Dataflow Computer for Scientific Computations, *Computer Physics Communications* 37/1 (1985) pp. 141-148.

[25] Yuba, T. *et al.*, The SIGMA-1 Dataflow Computer, *Internal Report* (ElectroTechnical Laboratory, 1987).

Current Status of JUNET

Jun Murai[†] and **Akira Kato**[‡]

Computer Centre
University of Tokyo
2-11-16, Yayoi, Bunkyo
Tokyo, 113 JAPAN

Department of Computer Science
Faculty of Engineering
Tokyo Institute of Technology
2-12-1, Ookayama, Meguro
Tokyo, 152 JAPAN

JUNET has been developed in order to provide a testing environment for studies of computer networking and distributed processing by connecting a large number of computers and by providing actual services for the users. Mechanisms to manage resource naming, Japanese character handling, and fast dial-up link using IP protocol have been developed for the network.

In this paper, the current status of JUNET focusing on the above topics are introduced. Brief description of native language supports on the network software is also described.

1 Introduction

JUNET[1] have been developed as the first attempt to establish a electronic communication environment in Japanese research and development communities. Text message exchanges such as electronic mails and electronic news have been provided as well as other advanced network services. Several international links to world academic networks have also been established. Since this is the only working environment for experimental studies on computer networking, various research topics including physical communication technologies, network protocols, network interconnections and distributed processing environments are in progress.

The purpose in the very first stage of the network, thus, was to provide an actual network services to researchers and put Japanese communities into worldwide academic networks. And then, we started to work to solve problems existing on the network such as Japanese character handling, name handling for distributed resources, and communication technologies.

JUNET started its operation in October 1984 connecting local area networks in major Japanese universities in Tokyo area[2] and it provides users with the means of the worldwide communications via various international links[3].

The domain addressing over UUCPNET[4] was introduced in May 1985 with a system to generate the address conversion sendmail[5] rules. High speed dial-up modes have been studied in order to increase the transmission rate of the communication. To achieve this purpose, UUCP enhancement with kernel driver development were done, and the efficiency using a dial-up line increased up to more than 13Kbps. This encouraged us to migrate to TCP/IP protocol suite[6][7] even on dial-up lines as well as leased lines. As the result of the development, the dial-up IP link is providing as high as 8Kbps in end-to-end transmission. With general IP transmission over high-speed leased lines and over X.25 public packet

[†] jun%u-tokyo.junet@relay.cs.net
[‡] kato%cs.titech.junet@relay.cs.net

switching network, this variety of links introduced us with immediate needs of advanced routing mechanisms which is one of our major field of studies at this point.

As for the internetworking between JUNET and other academic networks is concerned, two gateways are operating to exchanging electronic mails and news. One is Kokusai Denshin Denwa Co., Ltd. (KDD), an international telephone and telegraph company which serves a gateway between UUCPNET/USENETand JUNET, and University of Tokyo is providing a gateway function between CSNET[8] and JUNET which is the major path for most of the other academic networks in the world.

Generally, there are strong demands for Japanese character handling on computing environment and thus supports of Japanese characters in means of computer communications are one of the primary characteristics of JUNET. In order to provide the functions, a JUNET standard Kanji code was chosen and conversions between the network standard Kanji code and operating system Kanji codes are provided by the network application available for JUNET. The general computing environment for Japanese character handling are established as well in order to cooperate with the network environment. The statistics introduced in this paper shows drastic influences of Japanese character handling in computer network environment in Japan.

2 Profile of JUNET

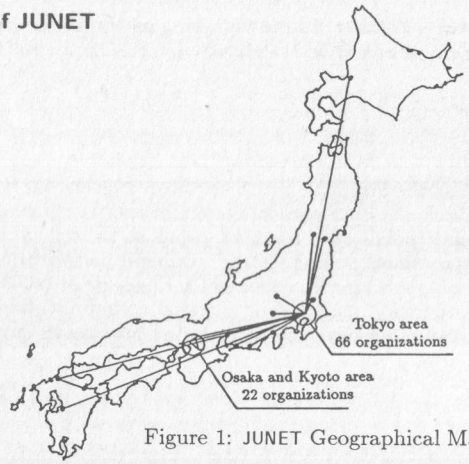


Figure 1: JUNET Geographical Map

JUNET currently connects more than 300 hosts in 106 organizations. The geographic areas covered by the network expands from Hokkaido, the northern island, to Kyushu, the southern island, however, concentrations in Tokyo and Osaka areas are obvious as shown in Figure 1.

Most of the links have been dial-up lines using 1200bps or 2400bps modems. although special mechanisms have developed for UNIX operating system and its communication software UUCP in order to use high speed dial-up modems with 9600bps or higher transmission rate. These mechanisms are also effective for TCP/IP protocols over dial-up telephone lines.

Organizations connecting to the network are universities as listed in Table 1, as well as research laboratories of computer software/hardware companies, and research laboratories of telephone companies whose domain names listed in Table 2. All the functions to operate the network have been administrated by administrators at each of the institutes in totally volunteer basis.

Table 1: Second level domains for universities in JUNET (Oct. 1987)

aoyama	Aoyama Gakuin Univ.	sheart	Univ. of Sacred Heart
hokudai	Hokkaido Univ.	shizuoka	Shizuoka Univ.
kansai-u	Kansai Univ.	sophia	Sophia Univ.
keio	Keio Univ.	titech	Tokyo Institute of Tech.
kit	Kyoto Inst. of Tech.	tohoku	Tohoku Univ.
kobe-u	Kobe Univ.	toyo	Toyo Univ.
konan-u	Konan Univ.	tsuda	Tsuda College
kyoto-su	Kyoto Sangyou Univ.	tsukuba	Tsukuba Univ.
kyoto-u	Kyoto Univ.	tut	Toyohashi Univ. of Tech.
kyushu-u	Kyushu Univ.	u-tokyo	Univ. of Tokyo
nagano	Nagano Univ.	uec	Univ. of Elect. and Comm.
nagoya-u	Nagoya Univ.	ulis	Univ. of Lib. and Sci.
oita-u	Oita Univ.	waseda	Waseda Univ.
osaka-u	Osaka Univ.	yamagata-u	Yamagata Univ.
osakac	Osaka Elec. and Comm. Univ.	yamanashi	Yamanashi Univ.
seikei	Seikei Univ.		

Table 2: Second level domains other than universities in JUNET (Oct. 1987)

adin	asahi	ascii	asr	astd	astec	atr
att-j	canon	canopus	casio	cec	citoh	csk
dcl	dec-j	decjrd	denken	dit	edr	etl
firmware	fujitsu	fujixerox	hitachi	hst	ibmtrl	icm
icot	ipa	jip	jsd	jus	jusoft	k3
kaba	kcs	kddlabs	kiic	kk	kubota	kyocera
m-giken	matsubo	meiosk	melco	mita	mri	nacsis
ncc	nec	nig	nts	ntt	oki	omron
pentel	ricoh	riken	roland	rrl	rtri	seclab
sharp	sharp-chiba	sigma	sony	sonytek	soum	sra
sun-j	toshiba	tytlabs	uclosk	univac	yhp	

2.1 Size of JUNET

Size of JUNET can be examined by various statistic values. Among them, the constant growth in number of organizations on the network is remarkable as shown in Figure 2. One or two new organizations are being connected to JUNET every week in average.

News articles are posted to the network constantly in the `fj` newsgroups which are currently distributed only within Japan[1]. The number of articles posted has been growing as in Figure 3, and 1807 articles are posted in October 1987, as the latest example. Note that only about 15 percent of the articles are written in English alphabet including articles in Romaji, an alphabetical representation of Japanese language, however, even among these articles most of them are re-posted messages from various mailing-lists. The major JUNET sites handles about 22 M bytes of articles in a month; 4 M bytes of `fj` newsgroups and 18 M bytes of USENET newsgroups.

One of the systems in the JUNET backbone handles about 200 M bytes of information in a month, and 85 percent of them are for the network news and 15 percent are for electronic mails. Since the network news are compressed to half of their size before the actual transmission, about 370 M bytes of text information is handled in one of the most busy systems in JUNET.

[1]Part of the `fj` newsgroups are distributed outside the country as requested.

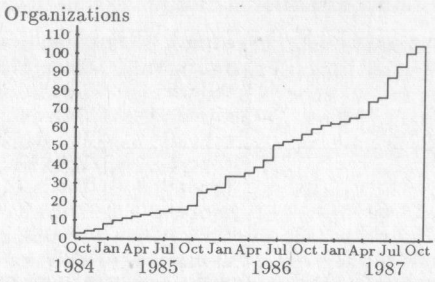


Figure 2: Number of domains

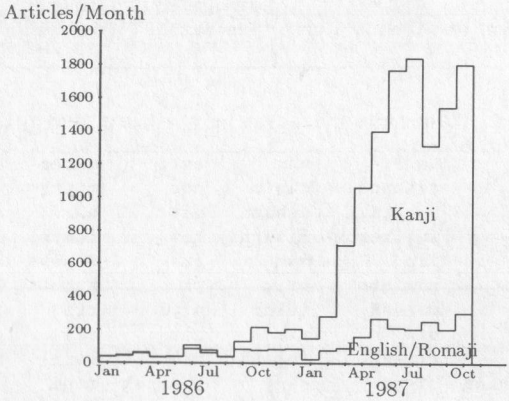


Figure 3: Number of Articles in `fj` newsgroups

The international information exchanges are served by two gateways of JUNET; one is in University of Tokyo and named `ccut.cc.u-tokyo.junet`. Another is `kddlab.kddlabs.junet` of KDD laboratories. The `ccut.cc.u-tokyo.junet` is on CSNET as `utokyo-relay` and is operated as the JUNET—CSNET gateway, on the other hand, the `kddlab.kddlabs.junet` is widely known as one of the backbone sites in the UUCPNET and is operated as the JUNET—UUCPNET/USENET gateway.

Table 3: International Message Traffics

gateway	mail	news
`kddlabs.junet`	10MB	18MB
`u-tokyo.junet`	13MB	0
total	23MB	18MB

The amount of international messages can be estimated by the total traffic at these two gateways. The example traffic of June 1987 are shown in Table 3.

In summary, JUNET currently has approximately 41 M bytes of international traffic in a month, and it has been increasing about 3 M bytes per month in last six months.

3 JUNET Domain Addressing

In the hierarchy of JUNET domain structure, a domain called '`junet`' is the top domain, although we are now preparing to employ ISO's country code (ISO 3116)[9] for Japan '`jp`' as the top domain name[10]. The second level domains are called sub-domains, and each of them represents a name of an institute or an organization. Lower level domains than the sub-domains may be determined at each of the sub-domains. In any cases, the lowest level domains are the names of hosts. The names of sub-domains usually are names well known to the society, but such names sometimes differ in intra/inter-national environment. Therefore, one or more names can be registered as synonyms for a sub-domain name to help users to address with general knowledge on the name of organizations. A name of a resource, thus, is defined in one of the domains.

There are one of the distributed name server in each of the domains which handles definitions and deletions of names using a database dedicated to that domain. A name server of a domain thus has a database to define names of lower level domains adjacent to the domain, or names of resources, such as names of mailboxes. The information held by each of the distributed name server is used in retrieving information of resource names and in accessing resources. By this concept, a resource can be defined in a logical domain; a mailbox can be defined even in the top domain, '`junet`'. This provides a name space which is well-matched to the naming concept of the real world yet providing consistency and efficiency of operations.

3.1 Design of JUNET message delivery system

There exists a name server for each domain where a distributed resource name can be defined. There is at least one name server in every hosts; a name server for a domain representing that host. Other than the one, name servers which represent domains located along a path from the root to this system in the domain tree can be existing in this system. Thus, name servers for the logical domains are managed by entities which are executed in a distributed manner.

A message delivery system using the above concept is implemented using sendmail[5] whose rule is generated by a rule generating system which plays a role of name servers of the JUNET naming concept.

Since the production rules of the sendmail system is differed site by site, the rules have to be generated at each site. To keep the consistency in the rules over JUNET sites, a configuration system to generate the necessary rules is designed and implemented. The configuration system to construct a name server receives information about domain names and about connections for communication among the name server. Then it generates sendmail rule as its output.

The domain database contains relations between the physical connections to the neighbor sites and the domain names which should be solved at that system. Other than the sendmail, the *rmail* command which receives messages through UUCP links was modified to handle JUNET addresses efficiently.

4 International Information Exchange

As described in this paper, number of messages exchanged in JUNET has been increased rapidly, and number of messages exchanged internationally via the two gateways have also been increased as well. Users of other networks, however, sometimes complains about the insufficient information on world networks generated form JUNET. The primary reason for the complains is obviously caused by the preference of Japanese characters among JUNET

users.

The development of general purpose software to handle Japanese characters in non-special hardware environment as well as development of the hand-made Kanji fonts encourages us to distribute JUNET Kanji messages to other countries. The experimental delivery of the domestic newsgroups to abroad has started in 1986 for some universities.

On the other hand, submission of news articles from USENET environment to JUNET can be achieved by adding a newsgroup called `fj.misc` to the newsgroup list of an article. This is efficient because all of the JUNET sites are subscribing `fj` newsgroups while some sites are not subscribing USENET newsgroups. The news article posted this way is handled as a JUNET news article inside Japan.

It is known that the addresses whose top domains are '`junet`' are properly handled at both `relay.cs.net` and `uunet.uu.net`. Therefore,

- *user*`%`*domain*`.junet@relay.cs.net`
- *user*`%`*domain*`.junet@uunet.uu.net`

are the most popular style of addresses from the ARPA Internet name space.

However, both international links are restricted links; mails to/from a JUNET user who does not register the addresses to the gateway can not be served. This is due to the cost of the international message exchanges. Furthermore, the mails to/from non-university users in JUNET can not be passed through the link between `ccut.cc.u-tokyo.junet` and `relay.cs.net`. Therefore, it is required to confirm that your friends in JUNET are registered in the gateways prior to send them your mails. Any questions should be mailed to the JUNET administrators, `junet-admin@junet`.

5 Domestic Language Support

One of the primary philosophies of JUNET developments is the pursuit of the better computer communications. For the purpose, the native language supports on the network environment is chosen as one of the goal. The statistics in Figure 3 shows that most of the news articles are written in Japanese characters, or in *Kanji* codes. It is observed that the large amount of electronic mails are also exchanged using Kanji codes. Therefore, one of the remarkable characteristics of JUNET among academic networks in the world is Kanji message handling for text message exchanges. In order to discuss about this topic, some basic ideas about Japanese character handling have to be discussed beforehand.

5.1 Kanji code

The history of Kanji code handling in computers is rather short, and there still are several confusions about Kanji code itself. That is to say, there are some different 'Kanji code standards' actually used in computer world. However, all the 'standards' refer to a single standard defined by JIS (Japanese Industrial Standard) X 0208 in their way of defining set of Kanji characters. JIS is always referred to because it can be introduced from any codes defined by International Standard Organization using ISO 2022 extension guidelines. It defines a Kanji code by two 7-bit bytes without using the most significant bits (MSBs). And there are several different Kanji codes exist for computer software because the switching method defined by ISO 2022 is not practical for random access to text messages.

The JIS X 0208 defines not only Kanji characters, but also English alphabets, digits, two types of 50 Kana characters (phonetic representation of Japanese language), and special characters. Among them, Kanji characters defined by JIS X 0208 are divided into two separated groups; one is called level one and another is called level two. The level one

includes about 3500 Kanji characters and this set provides sufficient number of Kanji characters for most of ordinary texts such as technical writings. There although need more complicated Kanji characters for the advanced text applications such that text including names of persons, names of places, or text of literatures. For the purposes like these, JIS defines level two Kanji character set including another 3400 characters.

In total, about 7 thousand Kanji characters are necessary for providing Japanese character capability in computer. Obviously, this requires more than one byte to represent, and representing a single character with two bytes is a standard concept. The important issue here is that we still need to use ASCII codes in computers as well as Kanji codes. This means we have to establish a way to handle a mixture of ASCII codes and Kanji codes. In order to solve the problem, there have to exists a way to distinguish ASCII codes sequences from Kanji code sequences.

In order to distinguish the Kanji sequences from sequences of ASCII representation of English alphabets and special characters, there are three major methods existing:

1. JIS X 0208 codes with JIS X 0202 (ISO 2022)
2. Extended Unix Code (EUC)
3. Microsoft Kanji Code (Shift-JIS)

By 1., a Kanji sequence is surrounded by a designating escape sequence of Kanji characters (`ESC-$-@` or `ESC-$-B`) and a designating sequence of English alphabets (`ESC-(-J` or `ESC-(-B`).

2. is originally defined by AT&T UNIX Pacific to provide internationalized version of UNIX operating system[11] and is becoming to be a standard way to represent Kanji codes in UNIX operating system in Japan. In the EUC, teh MSB's of both bytes in a single Kanji character are set to 1 whereas the MSB is cleared in an ASCII character.

3. was originally defined for CP/M on personal computers by a Japanese subsidiary of Microsoft Corporation. This is now a defacto standard for personal computers and is also supported in some of the Japanized UNIX environments. In the Microsoft Kanji codes, Kanji characters are mapped by some function so that first byte is in ranges of `0x81` – `0x9f` and `0xe0` – `0xff`.

In order to cooperate with the ISO standard method to handle character codes, and to utilize existing software which sometimes strips the MSBs off, we decided to use JIS X 0208 two 7 bit codes surrounded with escape sequences as the network standard. This decision requires conversion functions from JIS X 0208 to local operating system Kanji codes, namely EUC/JAPAN and Microsoft Kanji code.

5.1.1 Kanji code handling

In order to design network applications using Kanji codes, The following discussion have to be made regarding the average environments of existing JUNET systems.

1. There are several kinds of character codes including above three codes which are actually used in operating systems as internal codes to represent Japanese characters including Kanji characters. Among them, JIS X 0208 is not practical for internal code because of the complicated operations of switching modes when seeking a character in a byte sequence; random access to a sequence of byte is impossible. Thus, two 8 bit codes without escape sequence is preferable to two 7-bit codes with escape sequences as internal code for an operating system.
2. Since we have decided to use JIS X 0208 as the JUNET network standard Kanji code, we have to provide conversion mechanisms from JIS X 0208 to internal code of operating systems such as EUC and to Mircosoft Kanji code.

5.1.2 JUNET approaches

By assuming the above issues, we have developed the following environment for JUNET.

Kanji code: As we have described before, we are using JIS X 0208 with ISO 2022 / JIS X 0202 extension guide line as a network standard Kanji code. Escape sequence introducing JIS X 0208 can either be ESC-$-@ or ESC-$-B. These are two definitions of two minor versions of JIS X 0208 and both are legal in a sense of standard definition. Escape sequence introducing ASCII code is ESC-(-B and introducing ROMAN character code of JIS is ESC-(-J. Only a few characters are diffent between ASCII and ROMAN character set; \ and ~ in ASCII are replaced to ¥ and ¯ in ROMAN respectively. We therefore treat both of them equally. Note that the default code set for JUNET text message is ASCII code; there is no introducing escape sequence necessary if a text starts with a ASCII code.

Control characters: According to the ISO 2022 / JIS X 0202, any control characters appear in both Kanji sequence and ASCII sequence, however, deep backtracking may be necessary to determine the character mode when a file is accessed randomly. So we decided not to allow control characters appeared within Kanji code sequence; they can only appear in ASCII code sequence. This rule contributes to make software which handles text messages to be simple and transparent in terms of internationalizing because a function to find a control character can transparently be defined.

Single byte Kana code: There are another code set for representing Japanese character in a single byte; JIS X 0201. In this code, only Kana characters are represented. This code set used to be used in mainframes because Japanizing of software was easy. Since we now have Kanji code set anyway and all the Kana representations are included in JIS X 0208 using 2 bytes, we eliminate usage of JIS X 0201 to avoid the complexity caused from handling of three different code sets at a time.

Network software: In order to provide an environment for Japanese text message capabilities in JUNET, we have modified the following software to adopt the above strategies:

- Bnews[12] was modified to pass the escape sequences which used to be stripped off in the original version. In the standard implementation, the articles spooled are still represented in network code and the conversion functions to major internal Kanji codes are added. Other network news interfaces such as rn and vn have been modified, too.
- MH[13] has been added the code conversion facility between the network standard code and the local code.
- GNUemacs and MicroEmacs were modified to handle Kanji codes so that we can edit Japanese mails and news articles.
- X-window system was also modified to represent Kanji characters on both X.V10R3 and X.V10R4. Unfortunately, there were several different approaches to the modifications and offered us no interoperability. In order to establish an interoperability for X.V11, a group of researchers are organized with a mailing list in JUNET[14] for the purpose, and the recent discussions on the list is very active.

  Level one Kanji fonts have been *hand-made* and posted to JUNET so that a user can read and write Kanji characters without a special Kanji terminals.

An example to show the example JUNET environment of Japanese character handling is shown in Figure 4.

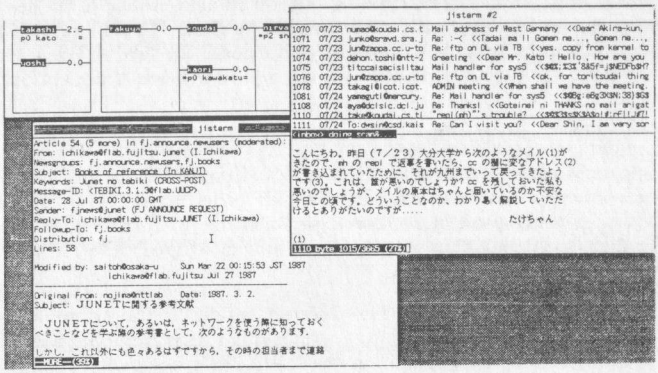


Figure 4: Example of JUNET Kanji Environment

6 Dial-up Links

6.1 Fast UUCP Links

The primary goal of the development of JUNET as the first step was to construct a network providing functions for text message exchange over research communities in Japan. Since JUNET is a volunteer project to provide basis of researches on network communication, UUCP protocol over dial-up links was our choice to start the network because of the popular, easy to set-up and inexpensive features of UUCP technology. The dial-up modems used have been upgraded from V.21 300bps to V.22bis 2400bps and we have almost succeeded to eliminate the V.21 and V.22 1200bps modems.

However, UUCP over 2400bps communication links is not practical as the size of the network has been grown and traffic has been increased. One of the problems is its slow speed, and it can be reduced by using the high speed modems such as Telebit's TrailBlazer or Microcom's AX/9624c. In order to make use of those high speed modems, the flow control mechanism between the modems and the hosts is required and *f-proto* UUCP is well known for Xon/Xoff flow control which is one of the popular flow control mechanisms. It is good for X.25 links but is not efficient for the hish speed modems because f-proto UUCP does 7 bit encoding. Instead of f-proto UUCP, *j-proto* UUCP which assumes 8 bit transparent link with Xon/Xoff flow control has been developed. To reduce the overhead in the tty device driver, a new line discipline called UTTYDISC which handles only flow control is also developed. With the combination of *j-proto* UUCP and UTTYDISC, we can get more than 13Kbps of transmission rate using TrailBlazers or AX/9624c's. The advantages of new UUCP is not only its high performance but also reducation of the tty port occupation.

6.2 Dial-up Links — Motivation

The UUCP link is intended for file transfer in batch mode and the file systems at the destination machines are liable to overflow. This problem is serious; when a file system overflow, many news articles and even mails are lost. Therefore, reliable datalink protocols which replace UUCP efficiently are preferable.

As JUNET is a volunteer based network and as we do not have enough funds to get dedicated lines between organizations, we have to make use of the dial-up likes at least for several months. X.25 packet switching network is one of the candidates. We can pass

the IP datagrams over X.25 links as defined in [15], however, it is not practical for the small organizations and for the newly coming organizations because of its cost. Another candidate is to pass the IP datagrams over the dial-up links. Providing this facility, we can make the network more reliable without any additional hardware. We have developed a serial IP module for dial-up links, called DL, derived from Rick Adams' SLIP.

6.3 Dial-up IP Link — Structure

IP connections over a dial-up link have been usually considered for personal computers at home[16], however, they are still effective for easy construction of an internetworking among universities, especially on the achieved performance with the high-speed modems. The system for dial-up IP link thus was designed and developed on SUN OS version 3.2 which is compatible with 4.2BSD UNIX.

The system consists of three modules. The *dldriver* resides in the kernel and it receives and sends IP datagrams with the IP module which also exists in the kernel space. It has communication entities as a "special file" in the UNIX file space called `/dev/dlx`.

The major portion of the system, *dldaemon*, is a permanent user process which reads information from the *dldriver* and controls it through `/dev/dlx`. The *dldaemon* is invoked at a boot time and resides on the system permanently. As soon as the process is initiated, it registers a network interface (*if*) called dl*x* and sets up the routing information in the kernel space so that the IP module can pass IP fragments with addresses routed toward the link to the *dldriver*.

The *dldriver* includes a tty line discipline called `DLIPDISC`, which does character mapping to escape DC1/DC3 and other control characters. Another module called *dlattach*, which is a user command, is invoked at login time to pass the information about the serial port to the *dldaemon*. Figure 5 illustrates the structure of the system.

A connection established with the DL is a point-to-point connection between a master system which initiates the connection and a slave system. The following example of sequence shows an overview of the procedure and the functions of the system:

1. When an IP fragment is selected to the dl*x* interface by the routing function in the IP module, the *dldriver* passes the fragment to the attached tty line if the link has already been established. Otherwise, the *dldriver* requests the *dldaemon* to initiate dialing to the remote site.
2. The master *dldaemon* logs-in to the slave site and invokes the *dlattach* command.
3. The *dlattach* provides the *dldaemon* residing on the slave site with the information about the serial port and about the caller's name, as shown in the dotted line of

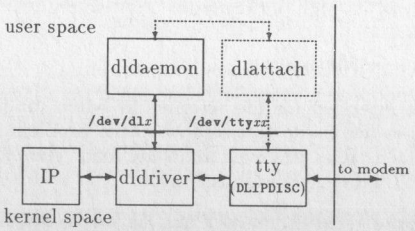


Figure 5: Structure of the DL

Figure 5.

4. Then, each of the *dldaemons* at the both systems issues a system call to switch a line discipline to `DLIPDISC`. Finally, they complete establishment of a dial-up link ready for IP communication between the systems.
5. A connection is terminated by the *dldaemon* when it detects that 60 seconds passed without any IP fragment transmissions.

The measured performance of the system with two TrailBlazers is approximately 7.5Kbps and two AX/9624c is approximately 8Kbps for end-to-end data transmission by ftp. Furthermore, we can obtain the performance of more than 10Kbps with an UDP based file transfer protocol whose window size is large enough to cover the delay originated from the internal protocol of the modems.

6.4 Discussions

The DL have the following characteristics compared to other media:

1. Link is usually dead and activated by requests.
2. Dialing takes 20 $\sim$ 60 seconds. The first datagram takes this long delay before reaching the destination.
3. Link speed is not so high. The propagation delay is relatively large because of the modems' internal protocol.
4. Link cost depends on the connection time rather than the amount of the traffic.

6.4.1 DL as a replacement of UUCP

The primary usage of DL is a replacement of UUCP. The messages are transferred in batched manner, however, each IP fragment is carried over the DL as in the dedicated lines. More precisely, `uucp` and `uux` replacements receive the file transfer requests and put the data once onto the spool directory. When the connection is successfully established, the processes which talk to the remote daemon with SMTP or NNTP are invoked. We have modified version of `nntpd` and a client process named as `sendnntp`, and we can prevent the file system from overflow.

The timing when the dial operation is initiated may be determined by an evaluation function depending on the amount of the data spooled for a particular destination, the elapsed time since last connection, and so on. The *dldaemon* executes such a function periodically, and determines whether it should initiate dialing.

6.4.2 Advanced usage of DL

For the DL as a replacement of UUCP, the existence of the link do not have be to inform to other sites. But the processes on the host other than the gateways want to send/receive the IP datagrams while the link is established. For this requirement, the gateways should propagate the routing information when the DL is established or closed.

Unlike the permanent network links, the DL is usually closed and initiated by the arrival of the IP datagram to the remote network. This means that the DL should be handled as 'active' in the routing information even the actual connection is not established. A process on a site other than gateway can issue a TCP connection request according to 'active' state in the routing information, however, it may time out because the dial operation may takes 20 $\sim$ 60 seconds. In such a case, the DL connection is established, while the TCP has been timed out and no actual data transmission is carried out.

We use the high speed modems for the DL to obtain good performance, however, those modems have relatively large delay time and we can not expect the maximum performance

with the ordinal TCP window size. Using a file transfer protocol with large window size over UDP, the performance of the DL exceeds 10Kbps, whereas 8Kbps with ftp.

In order to solve these problems and achieve the best use of the DL, a new routing procedure and congestion control technique suitable for the DL is now developing. The routing information should contain whether the path to a destination includes the DL or not. The time out mechanism and the window size detemination mechanism in the TCP module are affected to such a information in the routing information. Apart from the efficiency problems, the researches on the security issue should also be worked to use the DL practically.

7 Conclusion

Development of JUNET started in October 1984. Since then various researches on computer networking and distributed environment have been actively done as the rapid growth in the size of the network. Among them, name management functions to construct a hierarchical domain name space, Japanese character handling, and communication technologies using the high-speed modems have been focused as primary research concerns.

The actual work for the addressing and routing of JUNET text messages is achieved by a name server concept and its implementation. This system receives control messages to specify the information about logical domain name, connections and methods to deliver messages, and generates a sendmail rule set. This software provides an environment where the logical naming definitions and physical routing issues are clearly separated so that reliability, efficiency, extensibility, and flexibility of communication in the network are simultaneously achieved.

Internationalization of computer software is one of the most important issues in JUNET and its communication software. The clear separation of network Kanji code and internal operating system code employed in JUNET software provides a transparent environment on Japanese character handling in computer networks. Availability of Japanese messages obviously encourages JUNET users very much to exchanges messages over the network.

A new UUCP protocol and tty driver enhancement are developed for the requirements of higher transmission rate over the dial-up lines. As the result, more than 13Kbps UUCP transmission rate is achieved. This encouraged us to migrate onto TCP/IP suite using dial-up lines as well as using leased lines. Performance of the implementation of the dial-up IP link is about 8Kbps which is practical enough to construct a distributed environment over a widely interconnected network environment.

Progresses of JUNET technologies discussed in this paper lead us to many topics of future studies such as:

- Enhancement of name servers which handles general distributed resources.
- Establishment of gateway technologies such as optimal routing strategies based upon constructions of IP-based network using the dial-up IP link and IP over leased lines.
- Supports of multi-media message exchanges enhances the environment with multi-language supports currently achieved.

The technologies developed in JUNET can generally be used to construct a network interconnection with inexpensive cost.

Acknowledgment

The authors would like to thank Youichi Shinoda, Keisuke Tanaka, and Hiroshi Tachibana for their efforts in developing JUNET software. The dial-up IP link was achieved as a result of discussions with members of the JUNET project, especially with Susumu Sano.

References

[1] Jun Murai and Akira Kato. Researches in Network Developement of JUNET. In *Proceedings of SIGCOMM '87 Workstop*, ACM, 1987.

[2] J. Murai and T. Asami. A network for research and developement communications in Japan — JUNET —. In *Proceedings of First Pacific Computer Communications Symposium*, 1985.

[3] J. S. Quaterman and J. C. Hoskins. Notable Computer Networks. *CACM*, 29(10), October 1986.

[4] D. A. Nowitz and M. E. Lesk. *A Dial-Up Network of UNIX Systems.* Technical Report, Bell Telephone Laboratories, August 1978.

[5] Elic Allman. Sendmail – An Interconnecting Mail Rerouter, Version 4.2. In *UNIX Programmer's Mannual, 4.2 Berkeley Software Distribution*, Univ. of Californa, Berkeley, 1983.

[6] J. Postel (ed.). Internet Protocol – DARPA Internet Program Protocol Specification. 1980. RFC 791.

[7] J. Postel (ed.). Transmission Control Protocol – DARPA Internet Program Protocol Specification. 1981. RFC 793.

[8] D. Comer. The computer science research network CSNET: A history and status report. *CACM*, 26(10), October 1983.

[9] ISO. *Codes for the Representation of Names of Countries.* Technical Report ISO-3116, International Standard Organization, 1981.

[10] J. Postel. Domain requirements. 1984. RFC 920.

[11] Hiromichi Kogure and Richard McGowan. A UNIX System V STREAMS TTY Implimentation for Multiple Language Processing. In *USENIX Summer Conference Processings*, USENIX, 1987.

[12] S. L. Emerson. USENET: A bulletin board for UNIX users. *BYTE*, September 1983.

[13] R. S. Gaines S. Borden and N. Z. Shapiro. *The MH Message Handling System: User's Manual.* Rand Corporation, 1979.

[14] Hiroshi Tachibana. PD kanji font (tools and ascii fonts). Network news posted to `fj.sources` as `<1676@rika.cs.titech.JUNET>`, July 1987.

[15] J. T. Korb. A Standard for the Transmission of IP Datagrams Over Public Data Networks. 1983. RFC 877.

[16] D. J. Farber, G. S. Delp, and T. M. Conte. A Thinwire Protocol for connecting personal computers to the INETERNET. 1984. RFC 914.

SELF-ORGANIZING NETWORKS

Michael Frankel and Nachum Shacham
Information Sciences and Technology Center
SRI International
333 Ravenswood Avenue
Menlo Park, California 94025

James E. Mathis
Network Systems Development
Apple Computer
20525 Mariani Avenue
Cupertino, California 95014

October 1987

Abstract

With the advent of inexpensive, distributed, powerful computers has come the need for network telecommunications that can transfer data between machines and between users and machines. Fortunately, during the past two decades, major advances have been achieved in telecommunications, ranging from basic improvements in hardware and software design to the creation of such fundamental architectural concepts as the Open System Interconnect (OSI) model developed by the International Standards Organization (ISO).

However, although significant progress has been made, one area in telecommunications still requires further development. Until now, we have generally designed protocols for complex telecommunication networks, founded on the assumption that these networks are static. This assumption facilitates the design of routing algorithms, algorithms for network management, and network topology control.

This assumption, however, imposes constraints on the end-user of such networks. Specifically, a user with a high-performance workstation should be able to easily move his/her machine from network to network. However, this movement cannot occur in most networks without first notifying system administrators, in order that appropriate changes in host tables are established and distributed–a process that can take days or weeks to complete. Similarly, many networks today are constrained as to how they can modify their operating parameters (topology, channel-transmission rates, data-flow rates, and alternate routing) to provide service under adverse conditions, such as network-node failures.

Note, though, that these constraints are not fundamental to network designs. They have come about primarily because it has been easier to implement "first-generation" networks under these "static" assumptions. However, current telecommunications technology has matured to the point where we can now strive toward developing "second-generation" networks, which we shall call in this paper "self-organizing networks."

Our approach to discussing these networks will be to begin with a description of an open system architecture and an example of a network that meets most of the attributes of self-organization. Using these introductory examples, presented in Section 2, we will proceed to discuss the generic issue, "binding," fundamental to self-organizing network designs. We show that if a network is to be self organizing, dynamic binding must be supported.

We will then describe research we are pursuing in developing architectures, algorithms, and protocols that permit networks to self-organize. The results of this research will be networks that permit flexible access and that are inherently robust.

1 BACKGROUND

In this paper, we will focus our attention on packet-switched, telecommunications networks. To facilitate the discussion, we will view complex structures of multiple networks as a network of networks, called an *internetworks*. Within a network, packet-switches are homogeneous, while in an internetwork they can be heterogeneous. Networks are bound to other networks by gateways, to form the internetwork; these gateways serve to route packets from sources to destinations not found on the same network. To illustrate this, Figure 1 shows such an internework.

The networks, gateways, and hosts in Figure 1 adhere to an open system architecture. The layered protocols supporting this architectures are presented in Figure 2. These protocols have been developed and extensively tested within the United States, and are examples of first-generation designs. Two particular protocols in this figure will be discussed in this paper. They are the Transmission Control Protocol (TCP)[1] and the Internetwork Protocol (IP)[2]. As shown later in this paper, these protocols need significant redesign if they are to permit self-organization of the internetwork shown in Figure 1.

An example of a second-generation network is the Packet-Radio Network [3]. This network has been under research and development since the mid 1970s and is leading the way toward understanding how to design network-level protocols that permit dynamic, flexible self-organization.

The PRNET is a broadcast network that provides area coverage simultaneously to fixed and mobile users. Within the network, the common element is the packet radio (PR). Each PR contains an omnidirectional antenna, a spread-spectrum radio unit, and a digital unit. The radio unit transmits and receives packets under control of the microprocessor in the digital unit. In a stand-alone configuration, the PR is termed a *repeater*, and when a subscriber is connected to the PR, it is called a *terminal node*. Connecting a specially programmed microcomputer to a PR makes the combination a *station*. The station contains processes for routing packets, running, diagnostics, and taking measurements.

The packet radio receiver uses post-detection integration that takes advantage of the multipath often found in ground-based mobile operations. The integration extends over about one-third of a bit time, thereby trading some spread-spectrum time-capture potential for multipath resistance. The data rate is 400 kbps–unless the noise or multipath environment become too severe, in which case a 100-kbps rate is automatically invoked instead.

For fail-safe operation and load sharing, protocols have been developed to permit multiple stations to operate within the same network. Stationless-mode protocols have also been developed to permit reduced-capacity operations, should all stations become inoperative. Network management is exercised by the station through the use of intranetwork packets that are exchanged between all network elements. In the PRNET these data packets vary in size, up to a maximum of 2000 bits.

Packets are transported through the network on a store-and-forward basis, using buffers within each PR and a hop-transport protocol between them. Packets for forwarding are broadcast from a node (PR), but are selectively addressed to a single PR identified in the packet header. The relay process proceeds until the destination PR is reached, at which time the packet is passed across an interface to an attached subscriber device (for example, an intelligent workstation). Acknowledgments are required on a hop-by-hop basis along the route. Whenever an acknowledgment is not received for a packet (for any of several reasons), the sending PR retransmits the packet, a process

that continues for a set number times. If a link is not established, a similar number of attempts are made to route the packet through alternative PRs and, failing that, the packet is discarded, to guard against deadlocks. Because retransmissions can lead to duplicate packets, duplicate filtering is also performed in each PR.

Network management includes procedures for initialization, routing, access control, and flow control. Initialization in the PRNET, including the addition or deletion of individual nodes, is automatic. The condition of all PRs and all links is frequently monitored, using special status reporting packets. These data, supplemented by PR measurement data, are collected at each radio and form the basis for making real-time routing decisions. This procedure applies to mobile terminals as well. Thus, if connectivity from a mobile terminal to a repeater deteriorates, the mobile terminal is transferred to another repeater by a method that is transparent to the user.

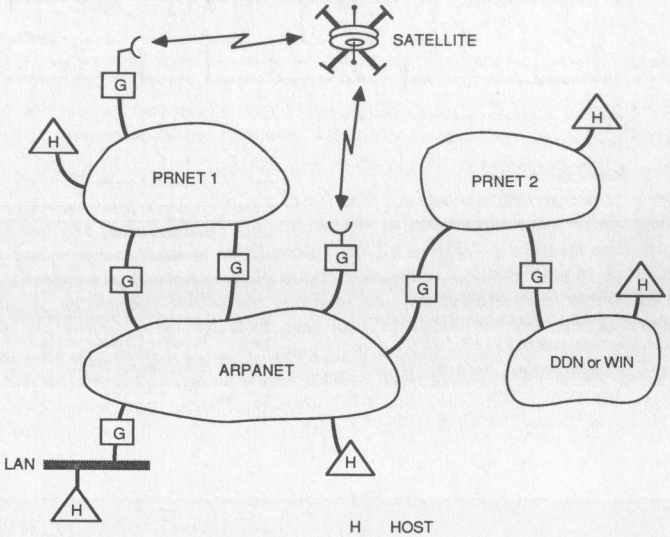
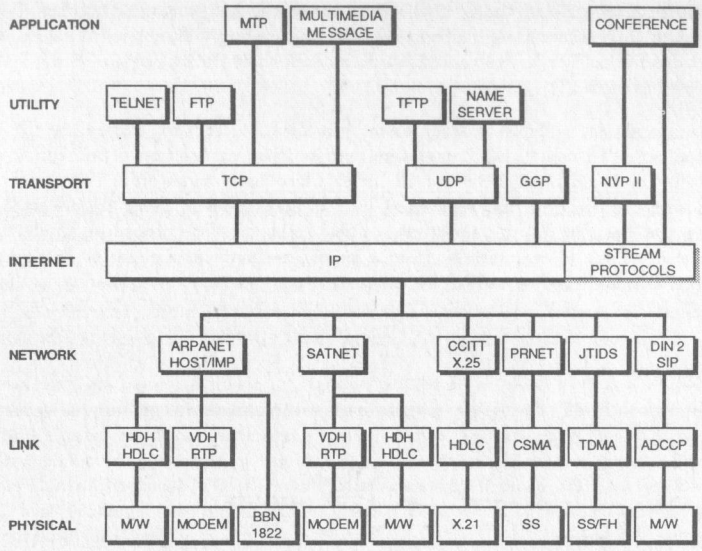


FIGURE 1 INTERNETWORK (network of networks)



ADCCP	Advanced Data Communication Control Procedures
BBN 1822	Spec for interface between host and ARPANET
CCITT	International Telegraph and Telephone Consultative Committee
CSMA	Carrier Sense Multiple Access protocol
DIN 2 SIP	Autodin II Segment Interface Protocol
FH	Frequency Hop
FTP	File Transfer Protocol
GGP	Gateway-to-Gateway Protocol
HDLC	High-level Data Link Control (procedure)
HDH	HDLC Distant Host
IMP	Interface Message Processor
IP	Internetwork Protocol
JTIDS	Joint Tactical Information Distribution System
MTP	Mail-TransportProtocol
M/W	Modem/Wire
NVP II	Network Voice Protocol
PRNET	Packet Radio Network
RTP	Real Time Protocol
SATNET	(Experimental) Satellite Network
SS	Spread Spectrum
TCP	Transmission Control Protocol
TDMA	(Synchronous) Time Division MultipleAccess
TELNET	Terminal/Host transport protocol
TFTP	Trivial File Transport Protocol
UDP	User Datagram Protocol
VDH	Very Distant Host
X.25	CCITT virtual circuit protocol

FIGURE 2 LAYERED PROTOCOL STRUCTURE

In the current PRNET, all users access the radio channel on the same frequency with the same spread-spectrum, psuedo-noise code. Access to the channel is controlled through protocols, called carrier-sense multiple access, to minimize packet collisions while network capacity is allocated to users based on dynamic demands made by their packet offer rate.

We will be using TCP/IP and the PRNET as the basis for much of the discussion that follows. We will begin by first addressing issues related to binding within networks and the internetwork. We will then address, in greater depth, self-organizing networks (using the PRNET as an example) and self-organizing internetworks (presenting extensions to TCP/IP), and finally, we will briefly discuss future research required to achieve truly self-organizing networks.

2 SELF-ORGANIZATION: BINDING

A required function within self-organizing networks is the use of automatic initialization and translation procedures within and between the various protocol layers. This initialization and translation process is part of the general problem of resource "binding," that is, the association of service requestor entities with service provider entities.

In networking, the term "binding" is most often used to describe the process of mapping the naming or address representation at a given layer to that used at the next lower layer. Thus, there is almost always a binding function performed at the service interface between two functional layers. While other types of parameter transformations are often performed at service interfaces, the name/address binding is unique, in that it is performed at virually every interface and forms a "thread" or series of transformations that start at the user/application interface and continue down to the hardware. Of course, more abstract forms of binding are found in such distributed-network technologies as distributed operating systems.

Shock[4] expresses the binding problem for networks as a translation from *name* (what) to *address* (where) to *route* (how). This characterization of the problem fits nicely into a layering of protocols. For example, the transport protocol is concerned with the identity (name) of the remote transport entity. This entity identification at the transport level is then translated into an address (or physical network port identifier) at the network level. Within the network, the address is translated into a route describing the process of transmitting the information to the destination network service point. Likewise, the application may be identified by a symbol string name which is translated into an address that identifies the transport service access point and an appropriate route.

There is always a design conflict between specifying a representation that is convenient for the service user and one that is efficient for the service provider. For example, in operating systems, textual file names are convenient for the user, but lists of disk block addresses are more efficient for the hardware; an important consideration in designing a file system is, therefore, to perform this mapping in an efficient and robust manner. The number of nested levels of binding also has an effect on the service provided, since each level of binding affords an opportunity to provide flexibility or hide information (or service complexities) from higher levels. For example, mapping logical tracks/sectors into physical tracks/sectors in disk controllers allows for the transparent handling of bad disk sectors. Therefore, as part of a structured design approach, the use of binding to hide information and complexity can contribute to a system implementation that is more easily adapted and suppported.

2.1 BINDING MECHANISMS

To understand binding mechansims, we first need to distinguish the process of establishing the initial binding (initialization) from that of mapping from one representation to another (translation). The initial bindings may be manually defined with automatic mapping, or the whole process may be automatic, requiring no manual intervention at all. Clearly, the less manual intervention in the binding establishment process, the more self-organizing the network will be. Extensive, manually-entered configuration information results in a network that cannot be quickly and easily installed by users and requires substantial configuration management upkeep.

A variety of techniques have been used to reduce the amount of manually entered configuration information used in today's networks. For example, in the Ethernet, the node address is specified as 48-bits to allow each manufacturer to automatically assign a globally unique indentifier (ID) to each board built. The burden is thus placed on the manufacturer, and users need not worry about ID assignments and potential address conflicts if two Ethernets are subsequently interconnected

by a bridge. In a similiar manner, the PRNET uses globally unique serial numbers assigned to subscriber devices to ensure no address conflicts should two PRNETs, operating on a common broadcast channel, merge.

A different approach is taken in the AppleTalk Personal Network to eliminate manual configuration information. Rather than assign an ID from a large address space at the time of manufacture, each device at system-startup dynamically picks an unused address using a broadcast negotiation scheme in which each node defends its address. While this scheme does not ensure global uniqueness, in practice it works well enough for small network with infrequent cabling changes.

As can be seen from these examples, the mechanisms used to accomplish binding and the processes by which the bindings are established are varied. However, there are some general aspects of binding mechanisms that can be identified. The simplest binding process is the *identity mapping*; that is, while the mechanisms to support a binding are present, the information is unchanged. The identity transformation must be distinguished from the situation where a binding is not made; that is, the mechanisms are not present. Other types of bindings include *algorithmic transformation*, *static table* look-up, and various dynamic or distributed procedures. In translations other than identity or algorithmic, the initialization process plays a major role in determining the degree of self-organization that the network displays. A network is hardly self-organizing if substantial efforts must be made in manually loading look-up tables.

2.2 SOME EXAMPLES

To fully understand the binding issues of a given network architecture, we must 1) identify the points in the protocol hierarchy where binding takes place, 2) determine how and when each binding is performed, and 3) understand the mechanism for establishing the bindings in the first place. To illustrate these concepts, we will look at various examples at the network-level, between the network and transport levels, and work through an application. While the specific examples are derived primarily from the TCP/IP protocol suite, the design principles are, nevertheless, applicable to other protocol families.

2.3 APPLICATION-LEVEL BINDINGS

Consider the bindings used in the TCP/IP electronic mail application as shown in Figure 3. At the highest level, the user's mailbox is identified by a symbolic string that defines the mailbox name (User) and "location" (Company.Com).[1] In this example, there are three main binding points: 1) from the mailbox location to the symbolic host name, 2) from the host name to the transport service entity's IP address, and 3) from the IP address to network address. A fourth binding point, from application to transport, is accomplished using a "well-known port" (WKP) or constant that identifies the application service with respect to the transport layer. A discussion of these binding points will illustrate some of the flexibility and limitations of the TCP/IP architecture.

The first binding, from the mailbox location to the name of the host holding the mailbox, is a function of the mail application and the domain-naming system. Originally, the mailbox location specified a physical host; with the adoption of the domain-naming system, the mailbox location can either be a physical host identifier, or the entire mailbox string can be interpreted as a subdomain identifier, allowing for generalized support for redirecting of mail and other mailing services. For example, the mailbox string "User@Company.Com" might be bound to the actual mailbox "U1012@Vax.Company.Com." The binding mechanism at this level can be used to suport a variety of electronic mail features, including mail distribution lists and mail forwarding. Furthermore, it

[1] In the TCP/IP suite, mailbox and host name translations are handled by the domain naming system [RFC 883] with names being hierarchical, read left to right, most to least specific. Thus "Company" is a member of the standard top-level domain ".Com."

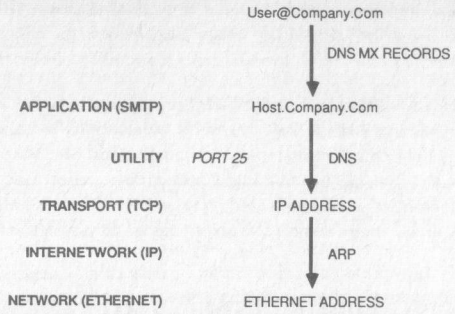


FIGURE 3 AN EXAMPLE OF BINDING

can also be iterative, allowing for several intermediate binding operations before resolving to an actual mailbox. A variety of such mailbox binding systems have been proposed or implemented, including the CSNET name server, the Xerox Clearinghouse, and the TCP/IP domain name system.

2.4 TRANPORT-LEVEL BINDINGS

Once an application or user has a symbolic host name, that name is translated into an address that identifies the transport entity. This binding allows for an invariant symbolic name independent of the physical internetwork address. The hierarchical structure of the naming space is completely separate from the structure of the internetwork address space and is managed independently. Unfortunately, one limitation of the TCP/IP model is the lack of a binding between the transport entity and the IP address. Normally, there is no need to separate the identification of the transport agent from the network module, and most existing protocols do not allow for such a separation.

However, in situations where the internetwork topology is very dynamic or the host is physically changing its network connection point, there arises a conflict between the desire for an invariant transport identifier and an internetwork address that provides for efficient route computation. Either the transport identifier changes, disrupting the protocol's operation, or the internetwork address cannot be used to compute routes directly, effectively breaking the binding at the next lower level. Within the TCP/IP model, we have developed an experimental extension to IP[5] that separates the TCP and IP addresses and provides support for dynamics in the internetwork topology (that is, changes in how the component network are interconnected). This work is described in more detail in Section 5, "Self-Organization: Internetwork Level."

2.5 NETWORK-INTERNAL BINDING

Various binding techniques are utilized within a network or in the interface between network nodes and subscribers to translate from subscriber names to network addresses. These translations are generally required when the name/address spaces are not one-to-one or such where features as logical addressing are desired.

For example, the address resolution protocol (ARP) is used by IP on Ethernets to translate from an IP host address, which is commonly 8 to 16 bits, to a 48-bit Ethernet interface address. Early attempts at using algorithmic transformations were cumbersome and led to a dynamic process in which each node maintains a cache of IP-to-Ethernet address pairs; when a translation is not known, a broadcast message is transmitted asking the machine with the indicated IP address to respond

with its Ethernet address. Thus each machine need only know its own IP and Ethernet addresses and can determine the Ethernet address of any other machine on its cable. This distributed lookup process is greatly simplified by the "free" broadcasting capability of the Ethernet.

The PRNET also provides a form of logical addressing as discussed earlier. However, broadcasting is not "free" in a packet-switched system with relatively low transmission speeds. In this networks, therefore, relatively complex information-distribution algorithms are used to efficiently distribute the address mapping to all network nodes. The distribution mechanism is tied to the process by which routing information is disseminated. In the PRNET, 1) all attached device addressing is logical, and 2) the mapping is one-to-one from the logical to physical address.

3 SELF-ORGANIZATION: LINK AND NETWORK LEVELS

Modern communication networks are complex distributed processing systems, consisting of geographically dispersed switches that control traffic access and are resposible for forwarding messages to their destination(s). The environment in which these networks operate exhibits dynamic behavior: traffic flow levels change frequently as connections are initiated/completed, nodes crash, and nodes are introducted into the network. In other cases, such as in the PRNET, links' qualities vary with time, as mobile nodes change their positions, and nodes leave and join the network, often at random times.

To function in such an environment, a network must acquire and maintain near-real-time information regarding the status of its links and topology, the offered traffic, and the well-being of its nodes. This information is used to regulate the data flow coming into the network, distribute the data for load leveling/congestion control, provide adequate performance, incorporate new nodes into the network, and route traffic around failed nodes.

In most networks, this information is collected by centralized controllers that are responsible for managing the network's operation. Centralized control, however, depends on the well-being of the controller, without which the network cannot function. For robustness, in the PRNET emphasis has shifted towards *distributed control* schemes, in which all nodes gather and maintain status information; they cooperate in managing the network. To do so, the nodes must acquire and maintain sufficient information to allow them to make decisions regarding the routing of data, selection of channel transmission parameters, the establishment of new links, and the downline loading of software to nodes that have failed. These decisions are made, of course, in consideration of other nodes and in cooperation with them. That is, the network is capable of self-organization in order to adapt to its changing environment; and the network's nodes actively participate in this process.

3.1 AUTOMATIC SELECTION OF TRANSMISSION PARAMETERS

In its present form, all the nodes of a PRNET share a single channel and utilize half-duplex transceivers and omnidirectional antennae for all transmissions and reception of data and control messages. This architecture allows mobile users to communicate without elaborate mechanisms for position location, antenna alignment, or transmission scheduling.

This architecture also gives the network several distinctive features: the common channel and omnidirectional antennae result in contention and collisions of overlapping messages. In addition, because of their geographical distribution, not all nodes are in direct radio connectivity with each other, a fact that makes resolving channel contention very complex.

Furthermore, PRNET topology is dynamic, because of variations in connectivity variation

caused by nodes' mobility and variations in channel signal-to-noise (SNR) rates. These variations are seen by the nodes as time-varying error bit/packet rates, and in extreme cases, as the breaking or reestablishment of links.

When the signal quality deteriorates, packets require more transmission attempts until they are successfully transmitted, which means that throughput decreases and the delay of packets travelling over a link is likely to increase. In the extreme case where the probability of packet success is so low that practically no packets can get through, links are considered to be disconnected.

Changes in the links' throughput increase the level of congestion and, because link throughput is a measure of its connectivity, large changes in the throughput are seen by the network-level protocols as topology changes. These charges are reported to other nodes so they can correct their routing tables, thereby increasing the network traffic load. If the changes are frequent, the delay in disseminating them causes the network to operate with inaccurate state information, resulting in suboptimal routing decisions.

In order to stabilize network performance under such conditions, the nodes must be able to adapt. The objective of the link-level protocols, therefore, is to mitigate the effects of these variations by modifying the network's transmission parameters. Such adaptation enhances the traffic-carrying capability of the network and presents to the higher protocol levels a more stable network, thereby reducing the amount of control traffic that has to be broadcast throughout the network.

There are several system parameters by which a node can improve the performance of the links to its neighboring nodes. In the PRNET these include

- Transmission bit rate
- Error-correcting code rate
- Transmission power.

It is relatively easy to provide the hardware capability in the radio for varying parameters[6]. The hardware, capability, however must be supplemented by the proper algorithms to control the selection of the parameter values. These algorithms must allow

- Channel conditions to be monitored
- The optimal parameter setting to be determined
- The setting between the transmitter and the receiver to be coordinated

The link between two packet radios can be observed at both the physical and the link-level: the receiver can measure the signal strength, noise statistics, number and distribution of errors in the packets, and the number of packets received with errors, as well as those received (or decoded) correctly. These measurements provide information on the bit SNR at the receiver's input and the channel throughput. They are, however, usually imperfect, noisy and fluctuate widely with time. Consequently, they are averaged to prevent frequent variation of the parameters' settings. With accurate measurements, the transmitter adapts its bit and code rates to deliver the best link throughput which is proportional to the product of the bit rate and the error rate.

3.2 ROUTING

Finding optimal paths in networks is a fundamental issue, and as such has been given considerable attention, resulting in a large variety of routing algorithms for packet-switched networks.

However, most of the development until now has focused on achieving maximum utilization of the network's resources under static or quasi-static conditions in which the network topology, its operating environment, and the offered-traffic pattern change relatively slowly.

However, in the PRNET mobile terminals introduce highly dynamic conditions; and as a result, the network must adapt quickly to changes. Routing algorithms, in particular, have to be able to acquire and maintain a path between any origin-destination pair even as these entities change their location in the network. Therefore as the network adapts, it must not induce

- Route loops
- Excessive message traffic
- Long blocking periods.

In this section, then, we discuss an adaptive routing algorithm we are investigating for use in dynamic networks, such as the PRNET.

3.3 SHORTEST-PATH ALGORITHMS

One of the most popular methods for routing is to forward all traffic between each origin-destination pair on the shortest path between them, where a path-distance metric is defined based on some cost function assigned to each link in the network. The various algorithms in this category differ in the way they assign cost to links, in the method of computing the shortest path, and in the procedure for updating the database(s) used for that computation after changes occur.

A simple implementation of the shortest-path algorithm assigns a cost value of "1" to all links in the network, resulting in paths of a minimum number of traversed links, or "hops." In a network where the traffic demands are well below the links' capacities, this method performs quite well. However, when the users' demand is more than negligible, the volume of traffic and its distribution in the network have to be taken into consideration to avoid congestion. One approach is to use a link cost which is based on the average packet delay over a link[7]. The idea here is to discourage sending traffic through highly congested links, by forcing the network to self-configure and choose the lower-cost, lightly loaded paths.

Naturally, the method by which the shortest path is computed is also important. Distributed algorithms, in which all the nodes participate in executing the algorithm, are the most applicable to self-organizing networks, because of their inherent fail-safe properties: there is no single node whose failure would cause the network to stop functioning.

The following have been the two most notable approaches for computing shortest path:

- *Partial information:* Each node keeps, for each of the N network nodes, a number that represents the value of the shortest path known to that node. By exchanging these distances with their neighbors, the nodes can iteratively compute the shortest path to all other nodes by executing the Bellman-Ford algorithm[7]. This scheme is used by the gateways in Internetwork.
- *Complete information:* Each node broadcasts to all other nodes in the network the list of its attached links and their costs, thereby providing each node with a full view of the network's topology. Using this database, each node can then execute any centralized shortest-path algorithm.

Unfortunately, neither of these schemes is completely satisfactory: the Bellman-Ford algorithm, for example, is known to cause route loops during its execution. Because it requires up to N iterations to converge, data may be misrouted in the network for long periods of time, thereby aggravating congestion in the network. This problem is particularly severe when nodes are deleted from the network, in which case the remaining nodes keep increasing their estimate of the distance to the deleted nodes ("the counting to infinity" problem), causing routing loops while doing so. Adding a distance upper-bound criterion to the algorithm causes it to declare a node unreachable as soon as the distance to it increases beyond some pre-specified value. This upper bound terminates the algorithm for a disconnected node, but since that bound has to be high to prevent unnecessary network partitioning, the number of iterations required to reach it becomes large.

The complete-information approach, on the other hand, requires large amounts of network-status data to be broadcast reliably from each node to all other nodes in the network. These data have to be sent for every change in network topology or link cost. As a result, when the network has a large diameter, the delay in delivering these data can be substantial.

A family of distributed shortest-path algorithms that possesses the attributes of the Bellman-Ford algorithm yet makes use of additional information to limit the route loops and reduce the time to convergence has been developed by Garcia-Luna, and is presented in a companion paper in this conference. Although we have implemented this new algorithm in a PRNET developed for the University of California, Los Angeles the performance of this protocol, under dynamic network topology changes, remains to be investigated.

3.4 QUASI-STATIC ROUTING ALGORITHMS FOR DYNAMIC NETWORKS

One disadvantage of single-path routing schemes, is their potential oscillatory behavior, which can manifest itself when a single traffic stream causes substantial changes to the delay in the links it passes through[7]. To avoid this problem, the network can maintain a set of paths from a given source to a destination and divide its traffic among these paths, a technique called *multipath routing*. Two main types of multipath routing are *quasi-static routing* and *dynamic routing*, both of which attempt to minimize average packet delay. Quasi-static routing is based on time-averaged link flows, whereas dynamic routing is based on current queue sizes in the network nodes.

We are presently examining using a relaxation algorithm as an optimization technique suitable for quasi-static routing since it lends itself to distributed operation[8]. In the relaxation algorithm, a node, i, maintains a number, called the *potential* p_i, along with the latest values of its neighbors' potentials as reported by them. Based on these numbers, node i can compute the values of the flow to and from each of its neighbors, based on the complementary-slackness conditions[8]. If this does not result in a feasible flow, a new potential is computed and a new iteration is carried out.

Because a feasible flow is not achieved until the algorithm converges, a node can use the intermediate, infeasible flow to decide on the fraction of traffic to send to each neighbor. However, the resulting flow may be infeasible or far from optimal.

We have therefore developed a modification to the relaxation method that reduces the time required to obtain a usable routing assignment following a link failure[9]. Using this new algorithm, node i still maintains the potential values p_i, but instead of computing the flow values, node i maintains routing variables that determine the fraction of the flow to be sent on each link. These flow values are computed by message exchange among neighbors. Based on these values, node i changes p_i and the routing variables. For a positive number δ, the algorithm continues until the complementary slackness conditions are achieved within δ.

If there is a path from every node to a destination d, the above algorithm converges and results in a feasible flow that is not optimal, because the complementary-slackness conditions are only

approximately satisfied. The parameter δ defines a neighborhood around the failed link outside which the algorithm will not be executed. Because the algorithm confines the message exchange to a small set of nodes around the failed link while leaving the flow outside that set unchanged, it converges faster than algorithms that require the participation of all nodes. Although the resulting flow is still suboptimal, δ can be chosen so as to achieve a flow arbitrarily close to optimal. There is a tradeoff between the distance from optimality and size of the set of participating nodes which determines the speed of convergence. By appropriately choosing the value of δ, the network can determine the proper balance between the two.

3.5 TOPOLOGY CONTROL

In the PRNET, a node is typically in radio range with only a subset of the total network nodes denoted as its *physical neighborhood*. Thus, the network employs store-and-forward techniques to transport packets over multi-hop routes. Packet forwarding decisions are based on routing tables, typically stored at the nodes. These nodes exchange messages with their neighbors to acquire and maintain the routing information.

The network must invest resources, such as channel bandwidth, processing, and storage, to acquire and update the routing tables. This overhead is proportional not only to the size of the network (the number of nodes in it) but also to the number of neighbors the nodes have (their *degree*).

The overhead in PRNETs with a high degree of connectivity may thus be substantial, reducing the network's efficiency. In networks with rich connectivity it may therefore be advantageous to limit the number of neighbors each node maintains full routing information about. That is, when a node has too many physical neighbors, it should maintain only a subset of its adjacent links for routing purposes. This makes the routing algorithm use only a subset of the physical topology, denoted here as the network's *routing* graph. Clearly, protocol that reduces the physical topology and provides a "reasonable" routing topology is needed for these situations.

We are interested in a *distributed algorithm* with which each node makes its own observations of its physical neighbors and participates in the decision-making process of topology reduction by exchanging coordination messages with its neighbors. The algorithm must satisfy constraints imposed by the operating environment. For example, in a distributed network it is hard to guarantee that all nodes will begin their operation at the same time. Thus, the algorithm must support changes and movement within the node population, accommodate late arrivals, and respond to the deletion of nodes from the network.

Because of the complexity in finding an efficient distributed algorithm that limits nodal degree while guaranteeing network connectivity, we have examined the performance of a very simple algorithm. The basic idea of the algorithm we have developed and are pursuing is to have each node select *at random* from its physical neighborhood a number of nodes, no more than its maximum allowed degree, and consider them to be its routing neighbors. This approach is appealing because it lends itself to a distributed implementation in which link are established by communication between neighbor nodes, and each node is responsible for not exceeding its maximum routing degree.

The protocol guarantees that the nodal degree is kept within specified bounds, and we expect that the routing network that results from the random selection will be globally connected with high probability. Random graphs, which are created by establishing a link between any two nodes with a probability p, have been shown to be connected with high probability for values of p above a certain threshold and to be disconnected with high probability below that threshold[10]. Although the routing graph is created in a different way than the "classical" random graph (for example, it has to satisfy nodal degree constraints, and not every node pair may be connected by a link), we expect that same threshold behavior to carry over.

We now describe the procedure for restricting the physical network. We assume that before the algorithm starts, all the network nodes are turned on, and each node is aware of its physical neighbors. This assumption is reasonable in a PRNET, for example, because the routing algorithm the nodes participate in require the nodes to periodically broadcast a packet declaring their presence.

Each node holds a list of its routing neighbors; that is, those neighbors with whom it has established a link that is considered to be a part of the routing graph. This list is initially empty. To add a physical neighbor to its list of routing neighbors, node i considers the list of routing neighbors already acquired. If their number is less than its maximum degree, node i selects one of the remaining physical neighbors at random, and initiates a handshake protocol that results in either a mutual agreement between the two nodes to include their link, or a rejection, which prevents the inclusion of that link. Once the handshake protocol terminates, nodes i and j either add each other to their routing neighbors list, or record each other's refusal to establish a link. Node i continues in this manner, and only terminates when it either achieves the limit of routing degree or has checked all of its physical neighbors. The protocol terminates when the last of the nodes terminates.

While executing the aforementioned protocol, the nodes participate in a routing table update protocol[11]. In this protocol, only the links of the routing graph are considered, and the resulting internodal distances represent paths completely within that graph.

If the aforementioned procedure results in a connected graph, the network can function according to the established routing tables. Otherwise, the nodes have to take corrective actions to make the routing graph connected.

If the physical graph is connected but the routing graph is not, some nodes observe physical neighbors that are unreachable by the routing table. That is, the nodes on the partition line are the only ones that can detect the partition. They will attempt to establish links to the disconnected parts, thereby reducing the number of partitions in the network. Again, this can be accomplished using a handshake protocol. There is, however, a potential difficulty in doing so: if one of the aforementioned nodes has already achieved its maximum degree, it must first disconnect a previously established link in order to reconnect the isolated node set. Notice, though, that the disconnected link may in turn cause another node set to be separated from the network, thereby creating a new problem, and possibly an oscillatory behavior of the protocol, in which nodes keep partitioning the network in their attempts to reconnect it. However, we do not expect this to pose a problem for networks with rich connectivity.

Another complication may arise when the nodes start executing the protocol at different times. This is expected to be more the rule than the exception, because in an operational radio network nodes are turned on gradually, and late entries into an operational network are quite common. The topology-restriction protocol causes a node to establish links up to the maximum possible. A late entry will thus find the existing nodes at their maximum connectivity, and establishing links will have to be done along with disconnecting others. To ease the entry of new links and to avoid having to simultaneously disconnect links while establishing new ones, the nodes will operate at steady state at a degree that is somewhat lower than the maximum possible. That is, nodes will leave room for new arrivals. Because of this, when a new node arrives, it can be immediately accommodated without the need for disconnecting existing links. This technique, however, will increase the degree of some nodes above the desired steady-state level. Thus, at some point such nodes will have to disconnect some of their links to return to the steady-state degree level, but this can be done without the pressure of having to accommodate a new node at the same time. Deleting of existing links to physical neighbors, required to keep the degree near the desired value, can be done either at random or according to some cost function assigned to the links. Notice that this provision allows the network to adapt to nodes' mobility as well.

We have simulated the operation of the topology-control algorithm described above. The simulation consisted of locating N nodes randomly on the unit square. The physical neighborhood was

specified by a number R such that a node's physical neighborhood was the set of nodes within a circle of radius R around it.

The physical network was first checked for global connectivity, and only the samples that constituted connected graphs were considered further. For each node, a logical neighborhood was selected according to the rules specified above, and at the end of the process, the global connectivity of the restricted logical network was calculated. For each set of parameter values (number of nodes N, maximum nodal degree M, and radius R) the process was repeated 100 times and the percentage of cases in which the logical network was connected was recorded.

The following four cases were simulated:

Case 1: All nodes are brought up simultaneously; the nodes are selected at random and the selected node adds one neighbor at a time.

Case 2: Nodes are brought up in two stages. First-stage nodes are allowed to complete the algorithm of Case 1, and then the second-stage nodes are brought up, forcing the former to delete links to accommodate them.

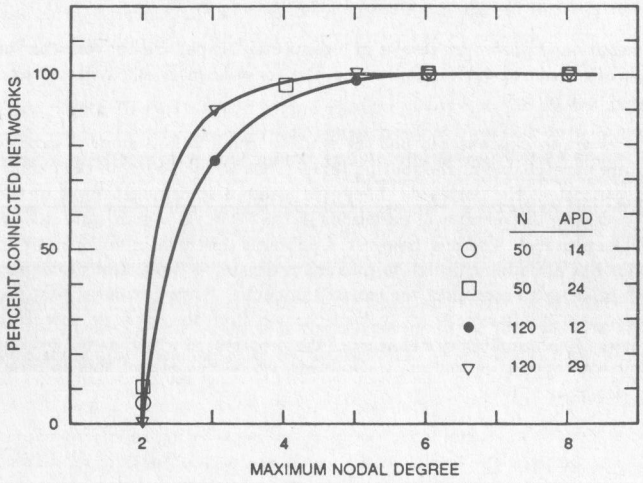


FIGURE 4 CASE 1

Case 3: Similar to Case 2, except that the network nodes are brought up one at a time.

Case 4: The network is brought up as in Case 1. A number of nodes are then moved to random new locations, and the algorithm is allowed to settle.

The simulation results for the four cases are presented in Figures 4, 5, 6, 7, and 8, respectively. The vertical axis is the percentage of cases that resulted in connected routing graphs; the horizontal axis is the maximum nodal degree per node (M) in the routing graph; N is the total number of nodes; and APD is the average physical degree of the nodes that resulted from a given connectivity distance R.

The figures depict an interesting behavior of the topology-control algorithm: the connectivity of the logical network as a function of the maximum nodal degree exhibits a threshold effect. If

we permit a maximum degree above the threshold, the resulting routing graph is connected with probability very close to 1. The threshold depends on the number of nodes in the network and on the average physical degree, but its value does not vary much: it remains between 3 and 7. This means that as long as the maximum permitted nodal degree is 7 or above, the resulting network is almost certain to be connected (for networks of 120 nodes or fewer). For nodal degree values below the threshold, the simple random-selection algorithm would not be sufficient, and a different approach should be taken.

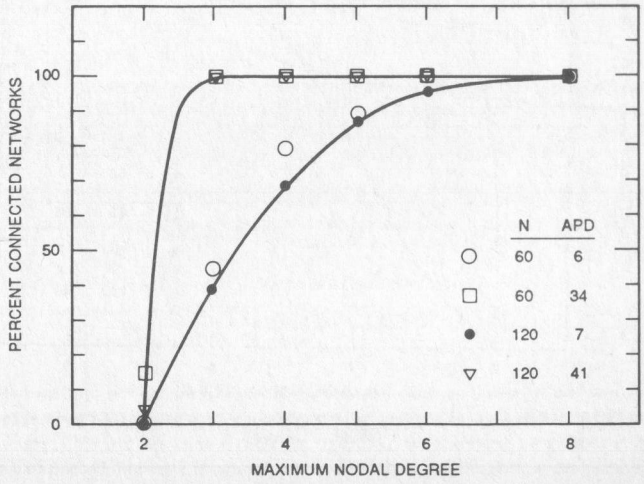


FIGURE 5 CASE 2

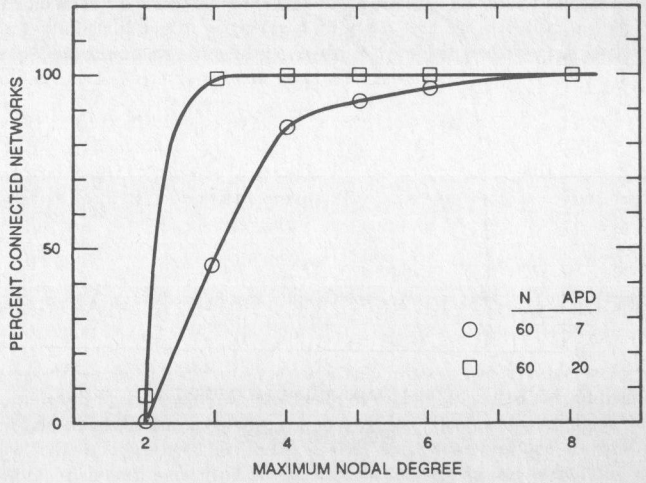


FIGURE 6 CASE 3

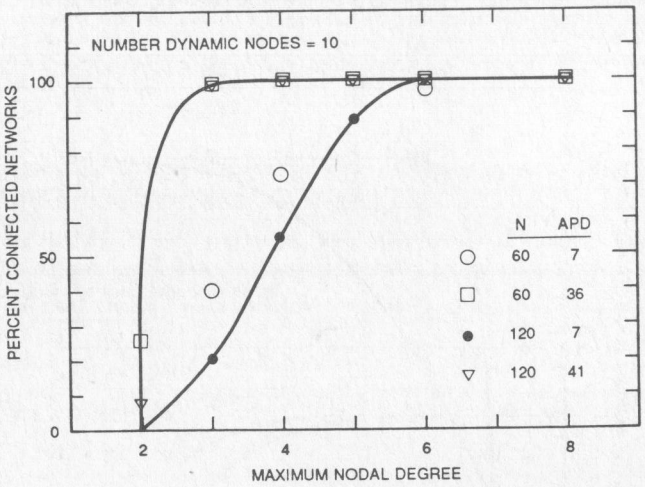


FIGURE 7 CASE 4

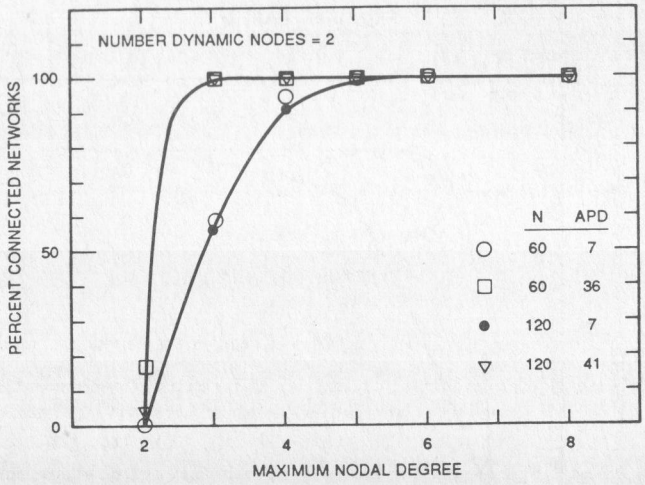


FIGURE 8 CASE 4

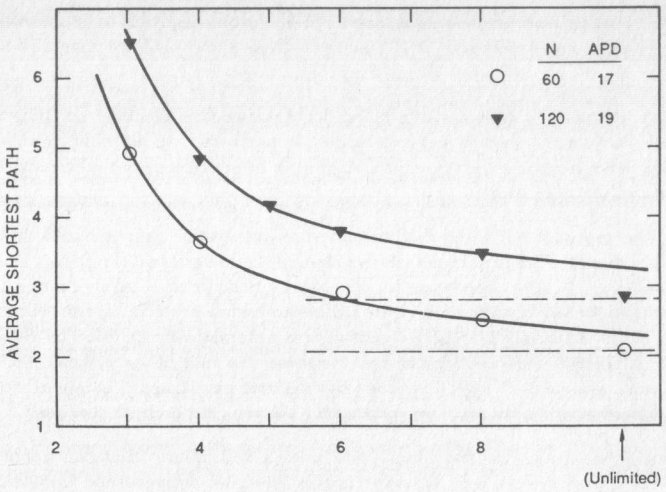


FIGURE 9 CASE 1

Another observation is that the threshold effect is much stronger when the APD (average physical degree) is about 20 nodes or more compared with the effect when the APD is less than 10. Stated another way, this implies that a random algorithm for selecting neighbors is more likely to result in a non-connected network in sparse networks (those within low average physical degree). One reason for this effect could be the following. In a sparse physical network, there is a greater likelihood that a node has only a single physical link to the remainder of network. If such a node exists, then the node at the other end of the single link must select this node as a neighbor to avoid a non-connected network. If this latter node has many physical neighbors to choose from (relative to the maximum number of logical neighbors permitted), and the selection is done on a random basis, then a non-connected network is likely. A similar problem occurs when two portions of the network are only connected by a very small number of links.

By limiting the number of routing neighbors per node, the average shortest-path length between two nodes in the network increases. A simulation was run on Case 1 that calculated the resulting average shortest path between all pairs of nodes in the network for different values of M (maximum nodal degree) and N (number of network nodes). The results of this simulation are presented in Figure 9.

4 SELF-ORGANIZATON: INTERNETWORK LEVEL

The internetwork described in Section 2 is essentially a static, two-level hierarchy with networks of hosts interconnected by gateways; this hierarchy is tightly bound to the addressing scheme of *net.host*, where *host* identifies the subscriber on network *net*. Conforming to the address structure, the routing also follows a two-level hierarchy, in that route computations are first performed to locate the network, then each network is responsible for delivery to the destination host.

Although the routing is not strictly required to conform to the addressing hierarchy, the internetwork systems components (gateways, hosts, and networks) lack protocols (hence information) to route other than along the address structure. Some protocols and experiments have been proposed

to decouple the routing from addressing to solve specific problems, such as expressway networks, mobile hosts, or partitioned networks[12,13]; however these protocols have generally not been implemented.

The routing structure is thus a static hierarchy, because it is identical to the static address structure, with its human-assigned network and host numbers. As a result, the deficiencies in the routing are derived from a tight coupling of routing to addressing (for efficiency) and a static address structure (for simplicity).

A static hierarchy does not mean that the internetwork system cannot handle link and nodes failures invisibly, because failures do not always change the hierarchical structure. But changes in the topological hierarchy, such as the merging of networks, must be reflected by changes in addresses; and these changes are not automatic and are visible to higher protocol layers. For example, the failure of one of the ARPANET-MILNET gateways is automatically handled by rerouting traffic through other ARPANET-MILNET gateways. However, the split of the original ARPANET into the MILNET and residual ARPANET (a very large network partitioning) happened smoothly only because of substantial planning and preparations for the required address changes.[2]

Self-organizing internetworks must cope with unexpected failures that can doom simple preplanned address change procedures. To address this issue, we designed and implemented an experimental version of a reconstitution protocol (RP). The RP system extends the internetwork architecture to keep the two-level hierarchy that binds addressing and routing for efficiency, but adds a dynamic addressing hierarchy. Because routing is easiest when performed according to some regular structure, the addresses are allowed to change as the overall topology of the internetwork changes or as hosts move from network to network to reflect current routing information.

With the addressing/routing hierarchy dynamic, the gateways can route traffic in a flexible manner to handle network reconstitution, though the dynamics are still not invisible. A further assumption at the transport protocol level (TCP) is that the connection end-point identifier at the transport level is identical to the address at the IP level. Thus any changes to the IP address, as a result of moving to a new network, for example, disrupt all existing transport-level connections to the host whose address has changed.

To provide invariant indicators for the transport level (required by TCP) while retaining flexibility at the IP address level, we introduced dynamic binding within hosts and gateways, so that the TCP identifier (name) stays constant while the IP address changes to reflect changes in topology.

Thus, the main technical approach to the internetwork reconstitution protocol centered around the following:

- Replacing the static two-level addressing/routing hierarchy with a dynamic hierarchy
- Adding a dynamic binding between the IP address and the TCP connection identifier
- Introducing a limited form of internetwork logical addressing.

These three key features are the basis of the RP work and were chosen for both backward-compatibility with existing TCP/IP implementations and efficient use of network resources.

[2]The planned addressing changes were a small part of the substantial effort required to plan and execute the trunking changes and node rehomings that constituted the actual network separation.

4.1 DYNAMIC ADDRESS/ROUTING HIERARCHY

An obvious approach to handling the problems brought about by changes in network organization is to eliminate completely the concept of a network and to have a flat addressing structure. Because the concept of a network would no longer exist, the problem of handling network partitions, network merging, and host movement between networks becomes moot. Unfortunately, the actual problem of how to route user traffic does not disappear; rather the amount of overhead traffic needed to exchange routing information increases dramatically and becomes unmanageable for large networks.

Current routing algorithms exchange an amount of information at least proportional to the number of nodes; introducing a flat address space would increase the number of nodes involved in the route computations and information exchanges from a few hundred to several tens-of-thousands. Thus, efficiency considerations dictate some form of hierarchical routing.

In homogeneous networks, such as the SURAN,[3] dynamic clustering (perhaps based on elections of cluster heads) is feasible. However, in the internetwork, gateway nodes have functionality (in terms of network connectivity) much different from host nodes; thus the hierarchy must obviously revolve around gateways. A clustering approach based instead on the individual packet-switches would not be practical, because of the substantial modifications required in all packet-switches.[4]

Thus the RP architecture is organized in a gateway-centric manner rather than in a network-centric manner. The gateway-centered approach immediately brings several advantages, the primary one being that gateways, as physical entities, do not partition; the gateway's interface to the network may fail, but this is a boundary condition at a network partition. Networks, on the other hand, being artificial entities, are subject to a variety of vagaries, such as partitions and mergers.

A further refinement of this model is to route based on gateway-halves; a "gateway-half" being associated with each network interface. While we use the term "gateway-half," the approach is trivially extended to cover gateways of arbitrary number of network connections (for instance, three-headed gateways), and we use the term in this generic manner.

Routing based on gateway-halves, or interfaces, has a number of advantages, including solving the triangle-routing problem, reducing the counting-to-infinity problem described previously, solving the ambiguity of how to address a gateway, and providing a unique way to identify groupings of hosts (as discussed below).

4.2 NAME/ADDRESS DECOUPLING

The second primary deficiency in the internetwork shown in Figure 1 is the close coupling of the IP address (for routing purposes) and TCP address (for connection identification purposes). In the original design of TCP, the use of the IP address was perfectly reasonable. IP addresses rarely changed—large mainframes seldom changed packet-switch ports on the ARPANET, the PRNET used logical addressing within the network to handle intranetwork mobility, and network number assignment was permanent. Moreover, the overhead of TCP/IP was considered high enough at the time without introducing an additional 64 bits of TCP source/destination connection identifier.

Thus, the TCP/IP addressing scheme cleverly collapsed the three elements of routing, addressing, and identifying (how, where, and what) into a single 32-bit field. Unfortunately, since connection identifiers must be invariant for the duration of a connection, the addressing and routing

[3] A new radio network technology for handling increased network size and survivability.

[4] The construction of a network comprising a variety of different transmission media is a separate topic of research. Here, we have constrained ourselves to considering the internetworking approach to heterogeneous transmission systems.

information must also remain static for the same duration. Thus, any changes to the address or routing indicators necessarily disrupt established transport connections.

The only solutions are to allow the reestablishment of transport connections (perhaps under control of a session protocol) or to allow the IP address to change. Since reestablishing transport connections would involve the loss of data (unless the session protocol had full error control duplicating the TCP functions!), the ability to change the IP addresses while maintaining TCP identifiers was included in the design of RP.

To maintain backward compatibility and allow for interoperation with normal TCP/IP hosts, the address changing is handled through an IP option. Specifically, at the source host's IP layer, old source and destination addresses are copied into an OLD-SOURCE-ADDRESS or OLD-DESTINATION-ADDRESS option, respectively, and the current IP address is placed in the IP header. In this way, the current destination address controls the routing of packets to the destination, and the current source address is available for the return of redirect messages. Once at the destination host's IP, the old addresses are copied back into the IP header (or equivalent data structure) before passing the packet to TCP or other higher-level protocols.

Another way to view these options is as TRANSPORT PROTOCOL SOURCE IDENTIFIER and TRANSPORT PROTOCOL DESTINATION IDENTIFIER options. If the identifier option is not present (the normal case), the identifier is taken to be equal to the IP address field, while if it is present, its value is used for the connection identifier.

The options, however, are implemented at the IP level rather than TCP for two reasons. First, the ability to change addresses while maintaining a transport connection may be of value to protocols other than TCP. Secondly, the gateway may be required to insert these options in the process of forwarding a packet to a host affiliated with a different gateway; to preserve layering, the option must therefore be at the IP level.

4.3 LIMITED LOGICAL ADDRESSING

Using a dynamic addressing hierarchy allows the gateway system to route around partitions and to handle merged networks by changing the address of hosts. The provision for separating the IP address from the TCP connection identifier allows a host to change its address while maintaining TCP connections intact. What is missing is a mechanism that quickly distributes the information about host address changes so that a transmitting host can correctly change the destination's IP address to track changes in network topology or host movement. The operation of this mechanism is further complicated by unpredictable addresses changes and by both hosts changing addresses simultaneously.

The RP architecture utilizes a two-level mechanism. For long response times, the domain name service must be able to change the address of host names registered in the database; a host would be responsible for updating its *name-to-address* translation entry using an appropriate protocol. The old address should be purged from the name system within 30 minutes of an address change. However, updating an address in the name server does not solve the problem for existing connections or connections established before the name database is updated (which may take many minutes in a large domain system with several servers).

To solve the short-term problems, we used a limited form of internetwork logical addressing. In this type of logical addressing, the old address for a host is dynamically rebound to redirect packets to the new host address. The service is limited, however, in that it is not intended to serve as a general logical-addressing mechanism that can be used to locate resources, but as a temporary "bridge" until the new host address is propagated and the use of the old address dies away.

When considering a logical-addressing mechanism, it is important to consider the replication of logical-physical-address mapping knowledge, how the database is maintained, and where the logical-to-physical address translation is actually performed.

In the ARPANET and PRNET logical addressing, the logical-physical address-mapping information is broadcast to every node in the network; usually this information is carried along with the route updates. The transformation to physical address is done in the node where the packet enters the network and the packet is then routed directly to the destination using the physical address. With the entire mapping database distributed to every node, this scheme is robust against node failures.

Other approaches concentrate the mapping function in server nodes. Here, the entry node must recognize and route the packet to the logical mapping server, which then translates the address. The servers must replicate the mapping database for reliability; but even so, there is no way to guarantee that every partition will contain a server.

The RP architecture adopts a compromise position. Because of the large numbers of gateways and hosts in the internetwork, distributing the logical-physical mapping information globally is infeasible. We instead concentrate the mapping function in the gateways "close" to the old location of the host. That is, the packet is routed towards the (now old) address; at a gateway close to the old destination address, the current address is put into the packet, which is then forwarded on. This approach provides for a natural distribution of the forwarding load, concentrates logical mapping information, and provides some level of redundancy.

In terms of functionality (without regard for performance), the RP protocols performed as expected, in that partitions and merges were handled automatically as a consequence of changing to a gateway-centric organization. The packet-forwarding functionality to handle network mobile hosts also worked as expected; however, this is a complex area involving trade-offs and scalability issues, which we have not yet had adequate opportunity to explore.

5 FUTURE RESEARCH

In a system currently being pursued by the authors, new network self-organization concepts are being explored. These include

- New organizational procedures that increase the intrinsic robustness of networks.
- Algorithms that allow a network to actively adapt its configuration to accomodate changes in traffic loads and patterns and failures in relay nodes.
- Extending the attention to congestion control up to the application, which may modify its behavior based on the available network capacity and delay.
- New network management and control schemes that attempt to bridge the gap between stated user goals and network algorithm parameters.

5.1 NEW ORGANIZATIONAL TECHNIQUES

One outcome of our self-organizing network research is the realization that the automatic organizational and management algorithms can themselves contribute significantly to the inherent robustness and flexibility of networks. In fact, self-organizing network algorithms may be the most cost-effective way to provide a substantial amount of network survivability.

For example, tests of the PRNET and simulations of other networks [14] have demonstrated the usefulness of the automatic alternate routing of data as an effective measure against some types of network failures. Another organizational technique for increasing robustness is based on incorporating into a single network a diversity of transmission media that would work in coordination to transfer user data. These so-called multiple transmission media/band networks have spawned a variety of studies that have shown that a network comprising multiple transmission media or frequency bands has superior reliability characteristics.

In our present thinking, the distinction between links, networks, and internetworks has been relaxed so as to treat the various transmission systems in a uniform manner. Thus, a *link* could be a point-to-point link (such as leased lines), a switched link (for instance, dial-up lines), a broadcast system (as in various type of multipoint radio), or an addressed system (a network, for example). These resources are viewed simply as communication mechanisms for transporting data between switching nodes.

To provide for this uniform architectural treatment of the various types of media and networks, transmission media are classified generically into the following four groups: 1) dedicated point-to-point circuits, such as radio links, 2) switched point-to-point circuits, such as telephone circuits, 3) addressed multidrop or multi-node systems, such as, 4) broadcast systems, such as multi-access satellite channels. This classification is based on both the sharing aspects of the media and their switching or addressing aspects. We will be investigating general mechanismsn for managing each of the types of links.

5.2 ACTIVE TOPOLOGY CONTROL

In the earlier discussion of network routing algorithms, the physical situation of the network was taken as given. That is, the network would *adapt* only passively to changes in connectivity, making no attempt to *influence* its physical connectivity. The placement of nodes and the system topology therefore had to be planned in such a way as to make extra resources available for rerouting wherever a potential partition might occur. However, systems that can reconfigure their interconnecting links or relocate relay nodes would significantly increase their ability to recover from massive node/link failures (such as caused by disasters) and to adapt to changes in traffic loads and patterns.

In many networks that use point-to-point links, the links are switched further at a physical-circuit level that is invisible to the packet switch. For example, for terrestrial systems, commercial technologies, such as ISDN, will enable networks to reconfigure interconnecting links relatively quickly, thus freeing land-based, point-to-point networks from the need to relay messages over preassigned links. The same procedure applies to circuit-switched radio links, in which the selection of operating frequencies, power levels, and antenna directions can change the nodal-interconnection pattern. Extensions of mobile radio network technology will allow autonomous nodes, based on platforms that can be relocated quickly (such as aircraft), to alter the geographical coverage of the network automatically, thus providing additional paths and resources for network reconstitution.

Thus, today's networking technology can be used to create new self-organizing networks that are capable, on demand, of establishing new connections, topology, transmission schedules, and routing patterns in a distributed manner. Several network self-reconfiguration strategies have been investigated[15], including a static approach that attempts to repair only broken links and a dynamic approach that changes network topology according to traffic demand. Work is also under way to develop heuristic approaches to this problem that simultaneously solve a variety of network topology constraints and operational requirements.

Investigations on the use of multirouting (that is, routing the same packet through different paths) and collateral information (such as position and location, the velocity of moving platforms, path loss, and other such physical measurements) to improve network robustness are also being

investigated. It may be possible for future networks to anticipate failures by precomputing alternative paths that possess highly uncorrelated failure modes and vulnerabilities. These precomputed paths would be used to send duplicates of critical, time-sensitive information as an alternative to flooding. The process of assessing vulnerability correlations in near real time is complex, however, and requires additional research on interpreting collateral information.

5.3 TRAFFIC-LOAD CONTROL

In an extension of traditional congestion-control techniques, our new research focuses on having applications software play a part in controlling congestion by varying its behavior. That is, it must actively manage the offered traffic load according to the network-supplied information about the approximate available capacity. This ensures that the most critical information will be distributed to remote sites, perhaps at the expense of not transmitting relatively less important details. For example, a distributed/replicated database application may elect not to update less important records to ensure sufficient communication capacity to maintain critical portions of the database in synchronization.

High packet-loss rates, for example, can cause transport connections to timeout or to retransmit packets excessively, thus causing further congestion. Congestion control must therefore be viewed as an integral part of the communication service offered and, as such, must be considered by the applications. The model we are currently pursuing is based on the following form of applications/network interaction:

- The network furnishes rough estimates of expected service (throughput, delay, and reliability) to the application.
- The application software utilizes the estimates of network capacity to adjust *dynamically* the precedence assigned to various components of the data stream, and perhaps even to the type of information transmitted or its encoding.
- The network uses this precedence to determine the order in which information is to be transmitted.

Here, the usual meaning of type-of-service (TOS) is extended to include factors other than requested delay or throughput. A variety of minimum TOS indicators is defined that, if not likely to be satisfied, indicate that the transmission of a data block should be abandoned. Each such data block could be considered as an atomic (that is, indivisible, basic, and independent) knowledge unit. Example service indicators include **Perishable**—deliver information within a given time interval; **Indivisible**—deliver all components of a multi-packet transmission; **Updated**—discard "old" packets buffered for transmission; and **Multicasted**—deliver to all multicast destinations. Several of these indicators would be attached to the data, and, together with priority and precedence information, would provide precise control of the queueing and transmission process by the higher software layers when the network is operating under conditions of severe resource depletion.

5.4 SYSTEM CONTROL

A final area of new research is in automatic system initialization and management protocols. These protocols take "high-level" direction and configuration-management information from a system-management operator and incorporate this information into routing and link control. This is not to say that an operator runs the networks directly, but rather that he exerts management control over network operations. The system manager will thus be able to control as traffic priority levels and access-control policies, enable and disable links or interfaces, and override certain routing and link protocol decisions.

A variety of management systems for packet-switched networks is under experimental development, including systems that can automatically monitor the network and detect faults. One such system, called the *signal officer's assistant*, uses artificial intelligence techniques to reason about optimal relay node deployments given user needs, time/resource constraints, and other logistic factors. These network-management systems provide the best of both the automatic and manual worlds—automatic data handling over networks whose traffic, routing, and topology control policies can be modified quickly and easily by users to meet changing mission requirements.

6 CONCLUSION

This paper has presented many ideas related to self-organizing networks and internetworks. We have shown that new ideas in algorithms for the design of such telecommunication systems can lead to completely self-managed, robust, and flexible systems. We, thus have named these networks "second-generation" to distinguish them from earlier systems developed under the assumption that the networks and their end-users were statically bound, and that the networks changed very slowly and predictably over time.

By relaxing these assumptions, networks and internetworks of the future will be able to change themselves to provide the best possible service under rapidly varying conditions.

REFERENCES

[1] Information Sciences Institute. *Transmisison Control Protocol.* January 1980. Prepared for Defense Advanced Research Projects Agency, Arlington, VA.

[2] Information Sciences Institute. *Internet Protocol.* January 1980. Prepared for Defense Advanced Research Projects Agency, Arlington, VA.

[3] *Proceedings of the IEEE, The Institute of Electrical and Electronics Engineers.* special issue on packet radio networks edition, January 1987.

[4] John F. Shock. Inter-network naming, addressing, and routing. In *Proceedings of 17th IEEE Computer Society International Conference*, pages 72–79, September 1978.

[5] Zaw-Sing Su and James E. Mathis. Internetwork accommodation of network dynamics: naming and addressing. In *Proceedings of International Conference of Computer Communications*, pages 681–685, October-November 1984.

[6] B. Willins and F.J. Bruno. Low cost packet radio characteristics. In *Proceedings of IEEE Globecom*, Atlanta, GA., November 1984.

[7] D. Bertsekas and G. Gallager. *Data Networks.* Prentice Hall, 1987.

[8] D. Bertsekas. Distributed asynchronous relaxation method for convex network flow problems. In *ORSA/TIMS Meeting*, November 1985.

[9] N. Shacham, J.J. Garcia-Luna, L. Aguilar, R. Ogier, and W. Zaumen. *Study on Distributed Processing System Topology For Battle Management Systems.* Technical Report, SRI International, September 1986.

[10] M.S. Chen. Shortest path in communication networks. In *Proceedings of IEEE Globecom*, Houston, TX, 1987.

[11] J. Jubin and J. Tornow. The darpa packet radio network protocols. *Proceedings of the IEEE*, 75(1):21–32, January 1987.

[12] Radia Perlman. *Flying Packet Radios and Network Partitions.* IEN 146, Bolt Beranek and Newman, June 1980.

[13] Jonathan B. Postel and Carl A. Sunshine. *Addressing Mobile Hosts in the ARPA Internet Environment.* IEN 135, Information Sciences Institute, University of Southern California, Marina del Rey, CA, March 1980.

[14] D.J. Baker and D.N. McGregor J.E. Wieselthier, A. Ephremides. Distributed network reconfiguration in response to jamming at hf. In *Proceedings of MILCOM 82*, October 1982.

[15] J.J. Garcia-Luna-Aceves and R. Rom. On the dynamic control of network connectivity. In *Proceedings of INFOCOM 87*, April 1987.

ROUTING MANAGEMENT IN VERY LARGE-SCALE NETWORKS*

J.J. GARCIA-LUNA-ACEVES

Information Sciences and Technology Center
SRI International
333 Ravenswood Avenue
Menlo Park, California 94025

Very large-scale networks and internetworks are already a reality, and routing constitutes an essential element for the operation of such networks. Unfortunately, the overhead of an adaptive routing algorithm becomes prohibitive in a network or an internetwork with several nodes (in the order of thousands or more) and a flat routing architecture in which each node must know how to reach every other node in the system. In this paper, we discuss the problems with existing network architectures and adaptive routing algorithms when they are applied to very large networks. Then we present a new hierarchical network architecture designed to solve these problems. The two key features of the new architecture are a new loop-free routing algorithm and a new hierarchical addressing mechanism. We compare this architecture with others previously proposed from the standpoint of the savings in routing overhead and the optimality of the paths obtained.

1 INTRODUCTION

As computing costs continue to drop, very large-scale computer networks and internetworks are becoming a reality. Even today, very large public and private computer networks and internetworks already exist [QUAR 86]. Routing is an essential element for the correct operation of these communication systems, and adaptive routing in particular is required to optimize the use of system resources. Store-and-forward switching is the technology that currently provides the most efficient way to utilize communication channels in computer networks [BERT 87]. Accordingly, we focus on adaptive routing in very large-scale packet-switching networks. However, we should point out that the addressing and routing mechanisms described in this paper can be applied to other types of computer communication networks.

In a routing algorithm that assumes a flat network architecture, each node must know how to get to every other node in the system, which means that the overhead fo maintaining correct routing tables becomes excessive when thousands of nodes are involved. If we look at the adaptive routing algorithms used in most of today's computer communication networks, we observe that they are variations of either the Bellman-Ford algorithm for shortest-path computation [FORD 62], which we call *BF algorithms*, or decentralized variation of Dijkstra's shortest-path algorithm (SPF) [DIJK 59]. The reader is referred to [SCHW 86] for an excellent survey of existing shortest-path routing algorithms and their performance. Unfortunately, both types of algorithms would incur excessive communicational, processing, or storage overhead in a very large-scale network. On the one hand, BF algorithms are too slow to converge after node failures and network partitions, because incorrect entries in the nodes' routing tables may form routing-table loops [GARC 88]. On the other hand, with a decentralized implementation of SPF, such as the new ARPANET routing algorithm

*This work was supported by SRI IR&D funds.

[MCQU 80], every node has to know the complete topology of the network. This means that an update message must be sent to all the nodes in the network every time the cost of a link changes. In a network with a great many nodes (on the order of thousands or more), it is clear that the communicational overhead becomes excessive. This problem has been discussed within the context of the DDN (Defense Data Network) by Seeger and Khanna [SEEG 86]. Furthermore, the storage overhead of maintaining a routing-table entry for each node in the network also becomes a challenge when some of the nodes of a network have storage limitations.

To alleviate the routing overhead problem of flat routing algorithms, the frequency of updates can be reduced; however, this inhibits the network's ability to adapt to topological changes, which may actually *increase* the communicational overhead because messages must be routed through very suboptimum paths and loops, and nodes may forward packets towards unavailable nodes. A more viable solution to this problem is *hierarchical routing*. The central concept in hierarchical routing is to employ a hierarchical addressing scheme in the network, to reduce the overhead associated with its routing.

Existing proposals for hierarchical routing attempt to reduce overhead by decreasing the amount of routing information each node has to maintain and communicate. They do so by organizing network nodes into *clusters* (also called *areas*), which are addressable entities that serve as the addresses of groups of nodes. However, decreasing the amount of routing information does not by itself reduce the communicational overhead created by a routing algorithm. Rather, the goals of a hierarchical routing algorithm should be to diminish the *frequency of updates* of addressing and routing information, while allowing the nodes to make routing decisions that are maximally close to the optimum. Another problem connected with existing proposals for hierarchical routing is that they are variations of routing algorithms that have already been implemented in flat networks, and are known to produce substantial routing overhead.

In this paper, we present a new approach to hierarchical routing that solves the aforementioned problems by employing a new distributed routing algorithm [GARC 88] and a new hierarchical addressing scheme. The new routing algorithm is loop-free and outperforms all previous loop-free, shortest-path routing algorithms. This algorithm, together with the new addressing scheme, reduces substantially the overhead of maintaining a routing hierarchy in a large network. The addressing scheme is based on making the name of a node (called *regional node*) the address of a group of other nodes (called *simple nodes*). In Section 2, we review previous approaches to routing in large computer networks with flat and hierarchical routing architectures. In Section 3, we describe the proposed hierarchical architecture in detail. In Section 4, we compare the performance of the new hierarchical routing algorithm with that of previous hierarchical routing algorithms; this comparison is based on the increase in path lengths experienced with the various routing algorithms, and on the overhead of the routing algorithms. In Section 5, we present our conclusions.

2 PREVIOUS WORK IN HIERARCHICAL ROUTING

2.1 Existing Proposals

The first significant work on hierarchical routing was done by McQuillan [MCQU 74], who proposed an extension to the old ARPANET routing algorithm (itself a variant of the Bellman-Ford algorithm) for hierarchical networks. This scheme was analyzed in detail by Kamoun and Kleinrock [KAMO 76]. According to McQuillan's proposal, the network nodes are organized into m levels of clusters, with each node belonging to exactly one cluster at each level of the hierarchy. Nodes represent clusters at level 0; a group of nodes forms a cluster at level 1, called 1-cluster; similarly, a group of clusters at level $k - 1$ forms a cluster at level k, called a k-cluster. The distance from a source node to a k-cluster represents the length of the shortest path from the source node to another node in the k-cluster. Each node maintains a routing table listing the shortest distance and next node to all

the other nodes in its 1-cluster, as well as to other i-clusters that belong to its $(i + 1)$-cluster. This same approach has been recently proposed for hierarchical routing for the ISO connectionless internet [ZAKO 85].

More recently, Baratz and Jaffe [BARA 83] have proposed a modification of McQuillan's hierarchical routing scheme to support virtual-circuit routing in a two-level hierarchical network. There, each node maintains an entry in its routing table for every other node in its same 1-cluster as well as for each boundary node in the network; a collaboration protocol is then used to obtain virtual circuits of optimal lengths between nodes in different clusters.

Meanwhile, Ramamoorthy and Tsai [RAMA 83] have extended the new ARPANET routing algorithm [MCQU 80] to work in a hierarchical network. In their scheme, nodes within the same 1-cluster use intracluster updates, according to the SPF algorithm, to obtain the shortest paths to one another. This implies that each node maintains the full topology of its 1-cluster in addition to its intracluster routing table, and that every update message is distributed to all the nodes in the 1-cluster. Boundary nodes are in charge of intercluster routing updates. A virtual network at level i is defined as one in which a link between two i-level boundary nodes is formed by the shortest path between them; the delay of the virtual link is then the delay in the path. The SPF algorithm is then used at every level of the hierarchy to allow each i-level boundary node to compute the shortest path to another i-level boundary node.

Besides these algorithms, as part of the DARPA-sponsored SURAN project, a number of hierarchical network architectures have been proposed and analyzed for large packet radio networks [GARC 85] [LAUE 86]. In these proposals, McQuillan's scheme is modified by defining boundary nodes and, in some proposals, special nodes (called global routing nodes or GRNs by Shacham et al. [GARC 85]) in charge of updating routing information at levels 2 and above. The algorithms for computing the shortest routes can include variations of the Bellman-Ford shortest-path algorithm. Another modification of McQuillan's proposal for hierarchical packet-radio networks is cluster overlapping, in other words, allowing nodes to belong to more than one cluster [SHAC 84] [GARC 85]; this is done to ensure the continuity of message delivery despite the nodal mobility of packet radio networks. Another difference introduced by some of these proposals[GARC 85] as well as McQuillan's, is that the distance from a node to a remote cluster need not be optimal, because the distance from a node to a cluster is measured by the number of GRNs traversed in the path.

Finally, Seeger and Khanna employ Ramamoorthy and Tsai's hierarchical scheme in a two-level network hierarchy designed to reduce routing overhead in the DDN [SEEG 86].

2.2 Limitations

It is important to note that, according to all of the existing hierarchical-routing algorithms summarized above, each network node has routing information on how to reach other nodes in its own cluster, and how to reach other clusters. In all of these algorithms, the address of a node becomes the name of a cluster in the network. In contrast, in a network with a flat routing architecture, the name of a network node constitutes its address, and a name-to-route mapping can be specified directly in the nodes' routing tables.

This results in an aspect of hierarchical routing that is often overlooked–namely, the fact that reducing the amount of routing information at each node creates an addressing problem, in that a node has to know where another node is located before it can route a message to it. For example, if node i wants to send a message to node j in another cluster, node i must first find out that the address of node j is cluster x; only then can it use its routing tables to route the message. The need for this intermediate name-to-address mapping in turn presents two important limitations.

First, in the general case, this mapping requires maintaining an address data base listing the name of every network node and its address (i.e., its affiliated cluster), and a cluster data base

listing the name of each cluster and a boundary node that can be used to reach it. To some extent, introducing address or cluster data bases defeats the hierarchical-routing goal of reducing the *amount* of information each node must have about other nodes, since this would require updating additional addressing information when nodes change their cluster affiliations or become inaccessible. And although a centralized data base could be used to reduce the nodes' storage overhead, it would be unacceptable, because the server maintaining the data base would be subject to too many queries and updates; the result would be a bottleneck for message forwarding. In fact, in a network that has to be highly survivable, or where datagram routing has to be done very quickly, the address and cluster data bases should be distributed to every network node.

To avoid the need for cluster and address databases, then, hierarchical addresses can specify the nodes' affiliation to clusters. For instance, node i in cluster C could be named "$C.i$", thus enabling every other node to know that the address of name $C.i$ is C. In turn, clusters can be defined according to the connectivity among nodes (that is, the network topology). This is the approach assumed in all of the existing hierarchical routing proposals. Its main advantage is that there is no need for tables containing the mapping of node names into cluster names as long as the affiliation of nodes to clusters is static. Unfortunately, this scheme fails in the event of topological changes, because of the need to dynamically maintain the affiliation of nodes to clusters, clusters to superclusters, and so on in the presence of topological changes. Even if all network nodes are static, a cluster can be partitioned due to link or node failures. And once a cluster is partitioned, a subset of the network may be unable to communicate with a remaining component of a cluster unless new names are assigned to the cluster component and its nodes.

The second limitation of existing hierarchical routing proposals is that they extend existing shortest-path routing algorithms based on the Bellman-Ford or Dijkstra's algorithms. As a result, they suffer from the same problems described elsewhere for such algorithms [GARC 88] [SCHW 86], although to a lesser extent. In particular, we note that all of the existing proposals for hierarchical routing employ routing algorithms in which routing-table loops may occur.

3 RNR–A NEW HIERARCHICAL ARCHITECTURE

From the foregoing, it is apparent that hierarchical routing does not ipso-facto reduce routing overhead. In fact, it can introduce considerable overhead due to cluster creation, partitioning, reconstitution, and node mobility. Furthermore, the routing algorithms currently in use present considerable limitations in terms of their time and communicational complexities, and (in some of them) looping problems. In this section, we present a new hierarchical routing architecture designed to cope with these limitations; we call it the *Regional Node Routing (RNR)* architecture.

3.1 Network Organization

According to the RNR architecture, the network nodes are organized into m hierarchical levels of regions. A *region* is simply a group of nodes, and each node belongs to exactly one region at each level of the hierarchy. A group of nodes forms a region at level 1, or 1-region; similarly, a group of $(k-1)$-regions forms a k-region. The regions defined in the network form a region hierarchy similar to the cluster organization of existing hierarchical routing proposals.

There are two types of nodes–simple and regional, and these are the only addressable entities. A *simple node* is one that every other node in its same 1-region knows how to reach. A *regional node* at level m (called an m-regional node) is one that every other node in the same $(m+1)$-region knows how to reach. Of course, a simple node can be considered a regional node at level 0.

3.2 Hierarchical Addressing

Nodes in the RNR architecture are addressed with respect to regional nodes and not the regions themselves. Each simple node is affiliated with one 1-regional node. Similarly, each k-regional node is affiliated with one $(k+1)$-regional node. Each k-regional node is assumed to always belong to the same k- and $(k+1)$-regions. Furthermore, each simple node belongs to only one 1-region.

From the standpoint of node i, the address of node j in its same k-region is the $(k-1)$-regional node to which node j is affiliated. If node j lies in the same 1-region as node i, node j's address is its own name.

In a network of m hierarchical levels, the name of a node in the RNR architecture is formed by concatenating m k-identifiers, where a k-identifier is formed by a k-region label and a k-regional-node label. A 0-identifier corresponds to a simple-node label. Notice that all k-regional nodes in the same k-region share the same k-region label; and, furthermore, the 1-regional-node label of a regional node is the same as its 0-identifier.

The label of a k-region is unique within its $(k+1)$-region in the region hierarchy and a k-regional-node label is unique within the k-region of the regional node. As a result, a k-identifier is unique within the naming scope of a $(k+1)$-identifier.

The concatenation of the $m-1$ i-region labels (where $1 \leq i \leq m$, and the m-region label refers to the whole network) and the 1-regional-node label of a node uniquely specify the 1-regional node to which the node is affiliated. The $m-1$ regional-node labels constitute a routing hint from the source to the forwarding nodes.

From the above, we observe that a k-region consists of the nodes that are affiliated with any one of the k-regional nodes that share the same k-region label. The key benefit of this approach is that all the algorithms required for the management of the routing hierarchy can be centered at the regional nodes. Because of space limitations, however, the focus of this paper is on the hierarchical-routing problem, that is, on maintaining consistent routing information in a network given that the addresses of network nodes (simple or regional) are known. In the rest of this paper, then, we assume the existence of algorithms that affiliate a node with a regional node and create regions by the regional nodes. The reader is referred to the work by Segall and others [SEGA 83] for a detailed description of network algorithms that can be used as building blocks of the mechanisms for the management of the routing hierarchy. The idea of using regional node names instead of cluster names as the basis for hierarchical addressing is very close in its nature to the gateway-centered addressing scheme first proposed by Su and Mathis [SU 85] for internetwork accommodation of network dynamics.

Figure 1 shows a network organized into regions according to RNR. In the figure, the k-identifiers of a node's name are separated by dots, and the k-region label is separated from the corresponding k-regional-node label by a hyphen. So, for example, the name 1-A.1-A.1, means that the node is affiliated with 2-regional node A of 2-region 1 and with 1-regional node A of 1-region 1.1, and its identifier within 1-region 1.1 is 1. Figure 2 meanwhile, shows a graph representing the affiliation of nodes to regional nodes, and Figure 3 illustrates the portion of the tree that is known to node 1-A.1-A.1. Notice that each node knows all the siblings (regional nodes at the same hierarchical level and descendants of the same regional node) of each of the nodes in the path to the top-level roots of the graph, and nothing else.

3.3 Routing Information

Each simple and regional node maintains a link table, a routing table, and a distance table. The link table contains the cost of all adjacent links. The cost of the link from node i to node k is denoted l_k^i. The routing table of node i is a column vector of at most $|N_i| + |R_i| - 1$ row entries,

where N_i is the set of nodes in node i's region, and R_i is the set of regional nodes known to node i. In addition to the node's identifier, the routing-table entry corresponding to node j consists of

$D_j^i(t)$: The length of the shortest path from node i to node j (a simple node in the same 1-region or a regional node) at time t. This entry is set to infinity when node i determines that node j is unreachable.

$N_j^i(t)$: The successor in the shortest path to node j at time t. This entry is *null* when node i determines that node j is unreachable.

The distance table of node i is based on the updates reported by its neighbors, and is a matrix with at most $(|N_i| + |R_i| - 1) \cdot |n_i|$ entries, where n_i is the set of active neighbors of node i. Entry (j, k), $k \in n_i$, in node i's distance table consists of the current shortest distance reported by node k to node j (denoted $D_{jk}^i(t)$), and a feasibility vector $(F_{jk}^i(t))$ that specifies the existence of outstanding acknowledgements to update messages sent out after node i freezes its successor to node j.

Routing information propagates outside 1-regions only if it refers to regional nodes. A k-regional node notifies routing-table changes that refer to l-regional nodes $(0 < l < k)$ to its neighbors, but nodes at the boundary between two regions stop the propagation of routing information that refers to simple nodes within each other's 1-regions. Within a 1-region, then, every node knows how to reach every other node through paths that only include nodes within the same 1-region. Within an i-region, every node knows which are the i-regional nodes in the region and how to reach them through paths that include only nodes inside the same i-region.

3.4 Packet Forwarding

To simplify our presentation, we assume the use of a 2-level hierarchical network in the rest of this section. The only purpose of regions in the proposed architecture is to reduce the frequency with which addressing and routing information needs to be updated. We have already stated that intraregion routing information does not propagate outside the regions. Also, because a node is affiliated with regional nodes and not the regions per se, packets can be delivered to their destinations even after a region is partitioned, and there is no need to redefine the node's address immediately after a region's partition.

The routing of packets between two nodes that lie within the same region occurs the same way as in a flat network. Each message contains the name of the destination node and the name of a regional node (called the *destination regional node*) that belongs to the same region as the destination node. Thus, packet forwarding is carried out as follows:

1. The source node determines whether the destination node belongs to its region or not.
2. If the destination node belongs to the source node's region, the source node looks up its routing table to determine if the destination is reachable. If it is, the source node obtains the next node towards the destination and forwards the packet accordingly. However, if the destination node is unreachable, the source node treats the destination as if it was in a remote region.
3. If the destination node belongs to a remote region, the source node looks up its routing table for the next node towards the destination regional node, which is the address of the destination node. If the destination's regional node is unreachable, the source node attempts to obtain an alternate regional node and forwards the packet accordingly. The alternate regional node must be a regional node with the shortest distance from the source node among all regional nodes whose 1-identifiers have the same 1-region label than the regional node specified by the source. If no alternate regional node can be found, the packet is discarded.

4. Each node in the path from the source to the destination does the same.

From the above, we observe that packets need not all go through regional nodes to get to their destinations. Furthermore, any node is reachable from any other node as long as it has a path to a regional node in its own region. Therefore, in contrast with existing hierarchical routing proposals, the delivery of packets to their destinations does not stop after a region partitions unless such destinations are unreachable from anywhere else in the network.

3.5 Routing-Table Updating

3.5.1 Main Concepts

A feasible successor to node j at time t for node i is a node q such that $D^i_{jq}(t) \leq D^i_{j\,N^i_j}(t-\epsilon)$, where $D^i_{j\,N^i_j}(t-\epsilon) < \infty$ is the value of its current successor's distance to node j just before time t, and $D^i_{jq}(t) + l^i_q(t) < \infty$. Also, the successor entry for node j is said to be in freeze state or frozen if $F^i_{jk}(t)$ is not empty for at least one neighbor node $k \in N_i$.

The goal of our algorithm is to make sure that, if and when all routing tables are correct and no messages are being processed or transmitted, D^i_j equals the shortest distance from node i to node j for any two nodes in the network. To achieve this, we extend the Jaffe-Moss routing algorithm using the concept of feasible successors defined above. In a nutshell, in terms of a node i's operation when it has to update its routing table, the algorithm is essentially the same as any BF algorithm as long as node i is able to find a feasible successor with the shortest distance to a destination j. Otherwise, node i freezes its successor to node j until it receives the acknowledgements from all its neighbors telling node i that they have incorporated its new distance to node j. However, node i may not have to freeze its successor to node j even if nodes in the path from N^i_j to node j have frozen their successor entries for node j.

3.5.2 Update Messages

There are two types of messages in our new routing-table update algorithm, *update messages* and *acknowledgements*. An update message is sent by a node to all its neighbors when the node must change its routing table. We assume that update messages are exchanged as soon as a topological change is detected. Each update message contains a message identifier (MID), a one-bit flag that specifies that the message is an update message ($ACK = 0$), and a list of tuples ($LIST$), corresponding to entries in the routing table of the node that is sending the update message. A tuple for node j in $LIST$ from node i consists of the identifier of the node, the shortest distance from node i to node j at time t, and a synchronization flag ($S^i_j(t)$) that is set to 1 when node i's successor to node j is in freeze state.

An acknowledgement to an update message is sent only to the neighbor node that sent the update message requesting an acknowledgement. It contains the same three components of an update message. In this case, however, $ACK = 1$, and the synchronization flag of each entry of $LIST$ must be set to 0.

An entry in the $LIST$ of an update message is simply called an *update*, while an entry of an acknowledgement is called an *ack*. When $S^i_j(t)$ is set to 1 in an *update*, it specifies that an *ack* is requested from the receiving node.

3.5.3 Update Algorithm

When a node is first supplied power, or resumes operation after a crash, it creates an empty distance table, a routing table with only an entry for the node itself, and a link table with the measurement of the cost of the active links adjacent to the node. Note that, after a node is initialized, it only knows its own identity and the cost of its adjacent links. After initializing itself, the node sends out an update message on each one of its active adjacent links to its neighbors.

Whenever a node detects that a link is established with another node, it sends an update message to that neighbor; the message contains the node's shortest distance to every other network node with the synchronization flags set to 0. Any node may enter or leave the network at any time without disrupting the operation of the routing algorithm.

For each network destination j, when at time t a node i determines that either (1) the cost of the link with its successor ($l^i_{N^i_j}$) changes its value, or (2) the link with N^i_j is gone (in which case $l^i_{N^i_j} = \infty$), node i updates its link table and determines whether or not its successor for node j, N^i_j, is frozen, i.e., if $F^i_{jq} \neq \emptyset$ for at least one $q \in N_i$. If N^i_j is not frozen, node i computes the new shortest distance D_{min} to node j, which is given by $D_{min} = Min\{D^i_{jq} + l^i_q \mid q \in N_i\}$.

After computing D_{min}, node i determines whether or not it has a feasible successor $k \in N_i$ such that $D^i_{jk} + l^i_k = D_{min}$. If such a successor exists, node i sets $D^i_j = D_{min}$ and $N^i_j = k$, and notifies *all* its neighbors of the new value of D^i_j with an update for node j in which $S^i_j(t) = 0$. This case corresponds to the typical operation of any BF algorithm, in which nodes update their routing tables with no internodal coordination. Otherwise, if node i cannot find a feasible successor with the shortest distance to node j, it freezes its successor entry to node j. More precisely, node i makes no change to N^i_j, sets $D^i_j = D^i_{j\,N^i_j} + l^i_{N^i_j}$, and sends an update for node j with $S^i_j(t) = 1$ to all its neighbors. Node i cannot change its successor to node j until it has received an *ack* to its update from each of its neighbors. An *ack* for entry j from node k to node i implies that node k has processed node i's update and has either found a feasible successor with the shortest distance to node j, or determined that it cannot reach node j. When node i receives the last of the *acks* from all its neighbors, it updates D^i_j, sends an *ack* to its current successor toward node j (if there is any) and an update with $S^i_j(t) = 0$ to the rest of its neighbors, and then updates its successor entry for node j.

On the other hand, if N^i_j is frozen, node i makes no change to N^i_j and sets $D^i_j = D^i_{j\,N^i_j} + l^i_{N^i_j}$. If node i's distance to node j increased from its previous value, node i sends an update in which $S^i_j(t) = 1$ to all its neighbors. Otherwise, the node i's distance to node j decreased from its previous value and node i sends an update in which $S^i_j(t) = 0$.

Any node k that receives an update from node i with $S^i_j(t) = 0$ carries out the same steps described above for node i after the detection of a link cost change or failure.

If a node k receives an update from node i with $S^i_j(t) = 1$ and $i \neq N^k_j$, then node k simply updates D^k_{ji} in its distance table and sends an *ack* to node i. On the other hand, if $k = N^i_j$, then node i carries out the same steps described above for node i.

It is important to node that node i can update its distance-table entry for node j and modify D^i_j after processing updates or detecting changes in $l^i_{N^i_j}$ while N^i_j is frozen. Also note that the decision of whether the successor to node j is frozen or not depends on the contents of the feasibility vectors. The handling of these vectors is described below.

3.5.4 Multiple Changes

Obviously, multiple increases of a node's distance to a destination may occur and acknowledgements to update messages sent from a node arrive from different neighbors at different rates. A node must know to which update a neighbor's acknowledgement is referring, and when it should propagate the acknowledgement to its own successor to node j. The proposed algorithm takes care of multiple link-cost increases using a mechanism that is almost the same as the one proposed by Jaffe and Moss for their algorithm [JAFF 82].

Every time that node i cannot find a feasible successor to node j after either (1) detecting an increase in $l^i_{N^i_j}$ or (2) receiving an update with the synchronization flag set to 0, node i adds a 1 to the end of F^i_{jk} for each $k \in N_i$. Similarly, every time that node i cannot find a feasible successor to node j after receiving an update with the synchronization flag set to 1 from its current successor, node i adds a 0 to the end of F^i_{jk} for each $k \in N_i$.

Because all updates and *acks* are processed in FIFO order at all nodes, when node i receives an *ack* from node k, it knows that this *ack* refers to the oldest update not yet acknowledged by node k. Accordingly, the leftmost bit is removed from F^i_{jk}. If node i receives an *ack* for node j from node k and $|F^i_{jk}| > |F^i_{jq}|$ for all $q \in N_i - k$, then node k is the last neighbor to acknowledge an update for node j. Node i sends an *ack* to its current successor to node j only if the leftmost bit of F^i_{jk} is 0. When F^i_{jk} becomes empty, node i sends an update with $S^i_j = 0$ to all its neighbors (except possibly its former successor, which already was sent an *ack*).

When link (i, k) fails while node i is waiting for *acks* from k, node i determines whether or not $k = N^i_j$. If k is the current successor to node j, then node i adds a 1 to the end of F^i_{jn} for all $n \in N_i$ to make sure that all its neighbors will send an ack for that event before it changes its successor, and then sets $F^i_{jk} = V^i_{jk}$, where V^i_{jk} is the unity vector such that $|F^i_{jk}| = |V^i_{jk}|$. The latter step ensures that node i will not attempt to send an acknowledgement to node k out of order in the event that the link with node k is re-established while node i is still waiting for *acks* from its neighbors.

If node k is not the current successor to node j, then node i carries out the following steps until it determines that $|F^i_{jk}| = 0$. First, node i determines whether $|F^i_{jk}| > |F^i_{jq}|$ for all $q \in N_i - k$; if that is the case, then node i knows that k was going to be the last neighbor to acknowledge one or more updates for node j. Accordingly, node i removes the leftmost bit from F^i_{jk}. If the removed bit is 0, then node i adds an entry in the *LIST* of the *ack* that is due to node i. When $F^i_{jk}|$ becomes empty, node i sends an *ack* to its current successor to node j with as many entries in *LIST* for node j as needed. If all feasibility vectors for node j are empty when node k's vector becomes empty, node i sends an update with $S^i_j = 0$ to all its neighbors (except possibly its former successor, which already was sent an *ack*).

The author [GARC 88] has shown that this algorithm converges in a finite time to the shortest distances after arbitrary topological changes, and that the paths specified by the nodes' routing tables are loop free at every instant of time.

4 COMPARATIVE PERFORMANCE ANALYSIS

In this section, we compare the RNR algorithm with the most representative algorithms for flat routing and hierarchical routing. In [GARC 88], the author shows that the new routing algorithm outperforms all previous loop-free routing algorithms. We thus consider only two flat routing algorithms: the old ARPANET routing algorithm, which is representative of BF algorithms, and the new ARPANET routing algorithm, which is representative of Dijkstra's shortest path algorithm (SPF). We likewise consider two hierarchical routing schemes, namely:

- A hierarchical version of the Bellman-Ford shortest-path algorithm, which we call the HF algorithm. The HF algorithm is essentially the hierarchical routing algorithm proposed by McQuillan [MCQU 74].
- A hierarchical version of Dijkstra's shortest-path algorithm, which we call the HD algorithm. The HD algorithm is essentially the hierarchical routing algorithm proposed by Ramamoorthy and Tsai [RAMA 83].

Our performance comparison is based on the communicational and processing overhead of the algorithms, and the quality of the routes obtained. We quantify the latter, as derived by means of a given routing algorithm, in terms of the ratio of the path length obtained in the worst case when this algorithm is used, to the length of the optimum path.

4.1 Routing Overhead

The overhead of a routing algorithm consists of the communicational overhead needed to update the nodes' routing tables after topological changes occur, and the storage and computational overhead needed to store and update the routing tables at each node. We measure the communicational overhead by the number of update messages that need to be exchanged, which we call *communicational complexity* (CC) of an algorithm, and the time it takes for the algorithm to update all routing tables correctly, which we call *temporal complexity* (TC) of an algorithm. To calculate the TC, we assume that the processing time at each node is negligible compared to the transmission delays. Therefore, we only measure the storage and computational overhead in terms of the *length* of the tables maintained at each node to support message routing, and simply call it the *storage complexity* (SC) of an algorithm.

In computing the SC, we do not include the overhead of maintaining an address table, as this would be very similar for the various hierarchical routing algorithms we consider and, furthermore, would depend on the characteristics of the algorithm used to affiliate nodes with regions or regional nodes.

To simplify our analysis of routing overhead and to provide a framework for comparing algorithms, we make two important assumptions. First, we assume a 2-level hierarchical network. Second, we assume that each algorithm behaves synchronously, such that every node in the network executes a *step* of the algorithm simultaneously at fixed points in time. At each step, a node receives and processes all messages transmitted by its neighbors during the previous step, and transmits its own message. The first step occurs when at least one node detects a topological change and issues update messages to its neighbors. During the last step, at least one node receives and processes updates from its neighbors, after which all routing and distance tables are correct and nodes stop transmitting update messages until a new topological change occurs. Note that the latter assumptions have already been used successfully in calculating the overhead of other distributed routing algorithms [JAFF 82] [GARC 85] [GARC 86] [GARC 88] [RAMA 83].

We consider two types of topological changes in our comparison. One type, which we call *intraregion change*, can make only those nodes that are in a single region change their routing tables. An example of the worst case of an intraregion change is the failure of a simple node. The other type, which we call *interregion change*, can induce changes in the routing tables of any network node. Examples of the worst case of an interregion change is the failure of a regional node or the partitioning of the network. In the rest of this section, we employ the following notation:

$|N|$: number of network nodes.
$|N_c|$: the average number of nodes in a region.
$|R|$: number of regional nodes in the network.
$|C|$: number of regions in the network, where $|C| < |R|$.

$|n|$: the average number of a network node's neighbors.
p : the average length of a shortest path between two nodes in the same region, measured in internodal hops.
d_c : the average diameter of a region, i.e., the longest shortest path between any two of its nodes.
d_n : the diameter of the entire network, measured in internodal hops.
d_b : the diameter of the virtual network of regional nodes.

The results presented in the next three subsections clearly demonstrate that the RNR algorithm outperforms previous routing algorithms for flat and hierarchical routing as regards routing overhead (i.e., the combination of TC, CC, and SC) incurred by the network.

4.1.1 Temporal Complexity (TC)

In the old ARPANET algorithm, $TC = O(|N|)$ in the worst case [JAFF 82]. In SPF, however, $TC = O(d_n)$ for any type of topological change (even a link cost decrease!), not just in the worst case [SCHW 86].

Now consider the HF algorithm. In the worst case of an intraregion change, $TC = O(|N_c|)$ [GARC 88]. In the worst case of an interregion change, though, $TC = O(|N|)$ [GARC 85] [RAMA 83], because of the looping problems of all BF algorithms.

Next, for the case of the HD algorithm, the temporal complexity after any type of intraregion change is $TC = O(d_c)$, which is identical to the temporal complexity of a network of diameter d_c running SPF. After any type of of interregion change, we have $TC = O(d_b \cdot p) + O(d_n)$ [RAMA 83]. There are two reasons for this: whenever an interregion change takes place, regional nodes have to distribute an update message in the network of regional nodes, which has diameter d_b and in which the average link length is p hops; and regional nodes must inform simple nodes of the result of updates made at the interregion level.

Yet, in the RNR algorithm, the worst case of an intraregion change yields $TC = O(d_c)$, while in the worst case of an interregion change yields $TC = O(d_n)$. These results follow directly from the construction of the RNR algorithm and from the result presented elsewhere by the author for flat networks [GARC 88].

Note that $d_n \leq d_b \cdot p$, which implies that TC is smaller in the RNR algorithm than in the HD algorithm. It is also clear that the RNR algorithm has a smaller TC than the other algorithms described here.

4.1.2 Communicational Complexity (CC)

In the worst case, the communicational complexity of the old ARPANET routing algorithm is $CC = O(|N|^2)$ [SCHW 86], which is also a result of the looping problems of BF algorithms. For SPF, on the other hand, $CC = O(|N| \cdot |n|)$ for every topological change.

For the HF algorithm, $CC = O(|N_c|^2)$ in the worst case of an intraregion change, and $CC = O(|N|^2)$ in the worst case of an interregion change. The latter is true because nodes participate as peers in updating routing tables and, in the worst case, a regional node may fail, an event that causes every node in the network to try all possible path lengths to that node until it reaches the value $|N|$. We observe, then, that the CC of this algorithm is smaller than in the old ARPANET routing algorithm, but larger than the CC of SPF for the worst case of an interregion change.

In the HD algorithm, $CC = O(|N_c| \cdot |n|)$ for every intraregion change [RAMA 83], and $CC = O(|R| \cdot |n| \cdot p) + O(|N_c| \cdot |n| \cdot |R|)$ for every interregion change [RAMA 83]. The latter results from adding the messages needed to disseminate the updates among regional nodes to those that are needed for the regional nodes to inform simple nodes of the changes made in their tables. We thus observe that the HD algorithm has a smaller CC than SPF only in the case of intraregion changes. When a change affects the connectivity of regional nodes, however, the HD algorithm has a larger CC than SPF. Notice too, that the HF algorithm may perform better than the HD algorithm, depending on the type of topological change involved.

For the RNR algorithm, meanwhile, in the worst case, $CC = O(|N_c| \cdot |n|)$ for an intraregion change, and $CC = O(|N| \cdot |n|)$ for an interregion change.

We note that $|N| \leq |N_c| \cdot |C| \leq |N_c| \cdot |R|$. Therefore, $O(|N| \cdot |n|) \leq O(|N_c| \cdot |n| \cdot |R|)$, and CC is clearly smaller in the RNR algorithm than in the HD algorithm. Furthermore, the RNR algorithm is consistently better than the old ARPANET routing algorithm and the HF algorithm. In the worst case, then, the RNR algorithm has the same CC as SPF, but has a smaller CC for intraregion changes. However, the RNR algorithm outperforms SPF in the average case, because updates from a node in the RNR algorithm propagate only to its neighbors, while every update has to be distributed to the entire network in SPF.

4.1.3 Storage Complexity (SC)

The storage complexity of the old ARPANET algorithm is $O(|N| \cdot n)$ if nodes maintain distance tables that contain the routing tables of a node's neighbors. In SPF, though, each node maintains a link delay matrix on the order of $O(|N|^2)$ and a routing table on the order of $O(|N|)$; therefore $SC = O(|N|^2)$ [RAMA 83] [GARC 88].

For the HF algorithm, each node maintains a routing table and a region table with the name of a regional node for each region in the network. Hence, $SC = O(|N_c| + |R|) + O(|C|) = O(|N_c| + |R|)$. And if distance tables were used in this algorithm, we would have $SC = O([\,|N_c| + |R|\,] \cdot |n|) + O(|C|) = O([\,|N_c| + |R|\,] \cdot |n|)$.

Next, in the HD algorithm, each regional node maintains [RAMA 83] an intraregion link-delay table, a routing table, and an interregion link-delay table. All this accounts for a CC that is larger than $O(|R|^2)$.

In the RNR algorithm, however, a regional node keeps a routing table, a distance table, which accounts for $CC = O([\,|N_c| + |R|\,] \cdot |n|)$.

From these results, it is clear that the HF algorithm has the smallest SC, and that the SC of the BC algorithm is not much larger than the one for the HF algorithm.

4.2 Quality of Path Lengths

Here we look at the relative *worst-case* increase in path length in networks that use the HF, HD, or the RNR algorithm, as compared with the corresponding optimum paths. We assume that links are bidirectional and that internodal distances are measured in number of hops. We use the term RNR paths to denote the paths obtained using the RNR algorithm. Also, we denote by r_{RNR} the ratio of the length of a RNR path between two nodes over the length of the corresponding shortest path; we use a similar notation for the HD and HF algorithms.

The RNR path between two nodes to the length of the corresponding shortest path is bounded by $r_{RNR} \leq 1 + d_c/2$, where $d_c \geq 1$ is the maximum diameter of a region in the network. To see this, consider two nodes located in the same region; the path P_{RNR} between them obtained

with the RNR algorithm can be as long as d_c, and includes only nodes in the same region. However, there may exist an optimum path between the two nodes that includes nodes in other regions; in the worst case, this path would include only one other node in another region and be, therefore, only 2 hops in length. Accordingly, we have that $r_{RNR} \leq d_c/2$.

Consider now two nodes a and b, located in different regions ($C(a)$ and $C(b)$, respectively). The length of the minimum RNR path obtained between a and b equals $w = |x_1| + |x_2|$, where $|x_1|$ is the length of the path from node a to a node n at the boundary of node b's region and $|x_2|$ is the length of the path from node n to node b. Similarly, the length of the shortest path between the same pair of nodes equals $w_{opt} = |y_1| + |y_2|$, where $|y_1|$ is the length of the path from node a to a node m at the boundary of node b's region, and $|y_2|$ is the length of the path from node m to node b.

According to the RNR algorithm, the path from any node to a regional node is optimum. Accordingly, if node b is a regional node, then $w = w_{opt}$ and the inequality of the theorem holds. Now assume that node b is a simple node, $w_{opt} \geq |x_1|$. Using this inequality it follows that $r = w/rw_{opt} \leq 1 + |x_2|/w_{opt}$. In the worst case, $|x_2|$ can be as long as d. Because node b is not a regional node, $|y_2| \geq 1$; furthermore, because a and b lie in different regions, $|y_1| \geq 1$, and the worst case satisfies $r_{RNR} \leq 1 + d_c/2$.

The three hierarchical algorithms that we are considering conceal the topology inside a region from the nodes outside that region. As a result, the three algorithms have the same worst-case increase in path lengths, as compared with the optimum paths, derived above for the case in which the source and the destination are in different 1-regions.

From the above, it follows that, in the worst case, the RNR algorithm performs at least as well as the HF and HD algorithms as regards the optimality of path length. Still, we should also point out that, in the general case, any hierarchical algorithm provides paths whose lengths can be very close to the optimum if the hierarchical path from a source to a destination includes the shortest path to the destination cluster or regional node, To see this, consider the RNR algorithm. A path from node a to node b in another region consists of a shortest path (P_{ac}) to a node c in node b's region, plus a path (P_{cb}) from c to b. On the other hand, the optimum path from a to b consists of a shortest path (P_{ad}) from a to a node d in node b's region, plus a shortest path (P_{db}) from d to b. On the average, P_{ac} and P_{ad} will have rather similar lengths, and the difference in length between P_{cb} and P_{db} can be just a few hops, especially if the average diameter of a region is short.

5 CONCLUSIONS

We have presented a new approach to hierarchical routing in very large networks, which we call RNR. We have shown that RNR effectively reduces the routing overhead in large networks and outperforms previous proposals for hierarchical routing in land-based networks. The two major reasons for the effectiveness of RNR are

- A hierarchical addressing scheme based on nodes well known by all the nodes within the same regions, which permits the design of simple algorithms for the dynamic maintenance of the routing hierarchy.
- A new distributed algorithm [GARC 88] for the calculation of shortest paths that provides the shortest paths when it converges after arbitrary topological changes, and provides loop-free paths even while the algorithm has not converged. The author shows elsewhere [GARC 88] that this new routing algorithm outperforms all previous loop-free, shortest-path routing algorithms for flat networks in terms of the storage capacity, number of steps, and number of messages required to maintain optimum paths.

Although our results show that the worst-case increase in path length relative to the optimum paths can be substantial, simulation results by Callon and Lauer [CALL 85] and Seeger and Khanna [SEEG 86] reveal that, on the average, the increase in path length due to hierarchical routing may not be very large.

References

[BARA 83] A.E. Baratz and J.M. Jaffe, "Establishing Virtual Circuits in Large Computer Networks," in *Proceedings of IEEE INFOCOM '83,* San Diego, California, April 1983, pp. 311-318.

[BERT 87] D. Bertsekas and R. Gallager, *Data Networks,* Prentice-Hall, Inc., 1987.

[CALL 85] A.R. Callon and A.G. Lauer, *Hierarchical Routing for Packet Radio Networks,* SRNTN 31 (5945), BBN Laboratories Inc., June 1985.

[DIJK 59] E.W. Dijkstra, "A Note on Two Problems in Connection with Graphs," Numer. Math. Vol. 1, pp. 269-271, 1959.

[FORD 62] L.R. Ford and D.R. Fulkerson, *Flows in Networks,* Princeton, New Jersey Princeton University Press, 1962.

[GARC 85] J.J. Garcia-Luna-Aceves and N. Shacham, "Analysis of Routing Strategies for Packet Radio Networks," in *Proceedings of IEEE INFOCOM '85,* Washington, D.C., March 1985, pp. 292-302.

[GARC 86] J.J. Garcia-Luna-Aceves, "Analysis of Routing Table Update Activity in Multihop Packet-Radio Networks," in *ICCC '86 Proceedings,* Munich, Germany, September 1986, pp. 648-653.

[GARC 87] J.J. Garcia-Luna-Aceves, "Regional Node Routing," Proceedings of IEEE Globecom '87, November 1987.

[GARC 88] J.J. Garcia-Luna-Aceves, "A Loop-Free, Shortest-Path Routing Algorithm," Proceedings of IEEE INFOCOM '88, March 1988.

[JAFF 82] J.M. Jaffe and F.M. Moss, "A Responsive Routing Algorithm for Computer Networks," in *IEEE Transactions on Communications,* Vol. COM-30, No. 7, July 1982, pp. 1758-1762.

[KAMO 76] F. Kamoun, *Design Considerations for Large Computer Communication Networks,* UCLA-ENG-7642, Computer Science Department, School of Engineering and Applied Science, University of California, Los Angeles, 1976.

[LAUE 86] G.S. Lauer, "Hierarchical Routing Design for SURAN," in *Proceedings of IEEE ICC '86,* Toronto, Canada, June 1986, pp. 4.2.1-4.2.10.

[JUBI 85] J. Jubin, "Current Packet Radio Network Protocols," in *IEEE INFOCOM '85 Proceedings,* Washington, DC, March 1985, pp. 86-92.

[MERL 79] P.M. Merlin and A. Segall, "A Failsafe Distributed Routing Protocol," in *IEEE Transactions on Communication,* Vol. COM-27, No. 9, September 1979. pp. 1280-1288.

[MCQU 74] J. McQuillan, *Adaptive Routing Algorithms for Distributed Computer Networks,* Bolt Beranek and Newman Inc., BBN Rep. 2831, May 1974.

[MCQU 80] J. McQuillan, et al., "The New Routing Algorithm for the ARPANET," in *IEEE Transactions on Communications,* Vol. COM-28, May 1980.

[QUAR 86] J.S. Quarterman and J.C. Hoskins, "Notable Computer Networks," in *Communications of the ACM,* Vol. 29, No. 10, October 1986, pp. 932-971.

[RAMA 83] C.V. Ramamoorthy and W. Tsai, "An Adaptive Hierarchical Routing Algorithm," in *Proceedings of IEEE COMPSAC '83,* Chicago, Illinois, November 1983, pp. 93-104.

[SHAC 84] N. Shacham, "Organization of Dynamic Radio Network by Overlapping Clusters: Architecture Considerations and Optimization," in *Proceedings PERFORMANCE '84,* Paris, France, December 1984.

[SCHW 86] M. Schwartz, *Telecommunication Networks: Protocols, Modeling and Analysis,* Menlo Park, CA: Addison-Wesley Publishing Co., Chapter 6, 1986.

[SEEG 86] J. Seeger and A. Khanna, "Reducing Routing Overhead in a Growing DDN," in *Proceedings MILCOM '86,* Monterey, California, October 1986, pp. 15.3.1-15.3.13.

[SEGA 83] A. Segall, "Distributed Network Protocols," in *IEEE Transactions on Information Theory,* Vol. IT-29, No. 1, January 1983, pp. 23-34.

[SU 85] Z. Su and J. Mathis, "Internetwork Accommodation of Network Dynamics: Organizational Structure," in *Proceedings IEEE INFOCOM '84,* Washington, D.C., March 1985, pp. 428-430.

[ZAKO 85] S.J. Zakon, "An Architecture for Routing in the ISO Connectionless Internet," *ACM CCR,* Vol. 15, No. 5, October/November 1985, pp. 10-39.

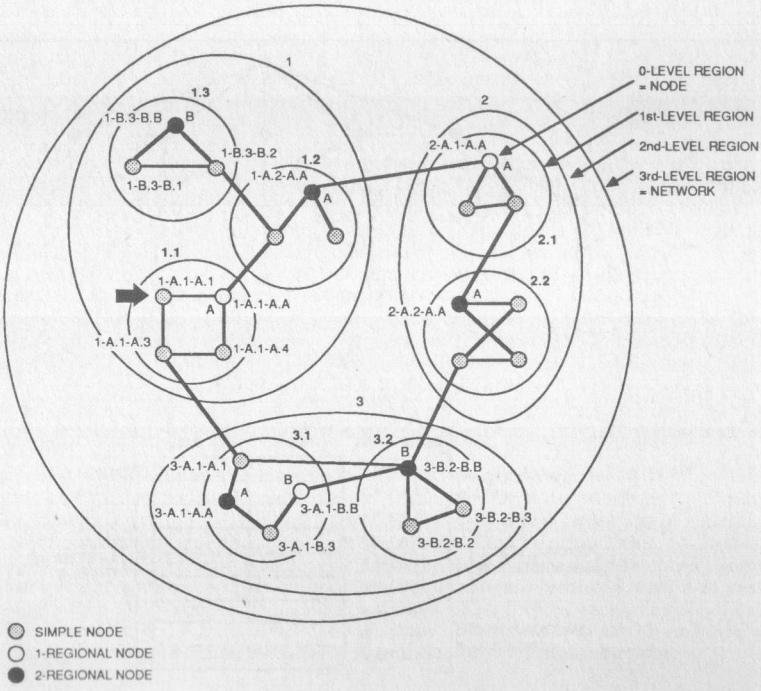


FIGURE 1 NETWORK ORGANIZED ACCORDING TO RNR

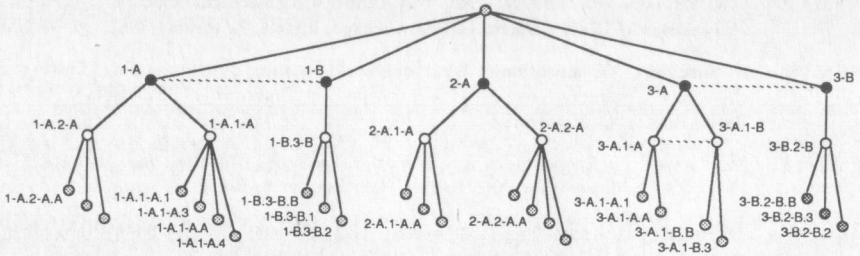
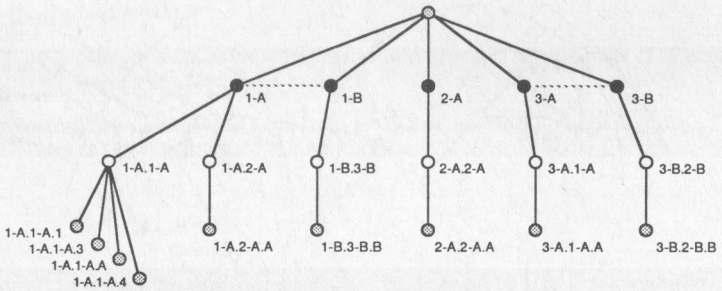


FIGURE 2 TREE REPRESENTATION OF RNR NAMING



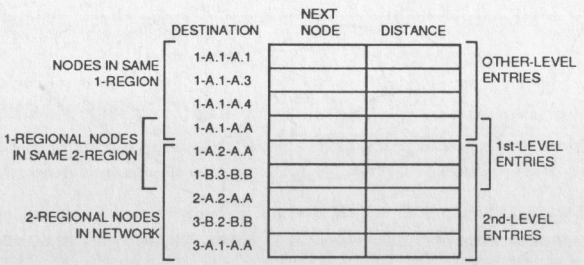


FIGURE 3 INFORMATION IN ROUTING TABLE OF NODE 1-A.1-A.1

3-D VLSI Technology in Japan and an example: a Syndrome Decoder for Double Error Correction

Takakazu Kurokawa and Hideo Aiso

Dept. of Electrical Engineering Keio University
3-14-1 Hiyoshi, Yokohama 223 Japan

Abstract

This paper consists of two parts. The first part is a survey of Three-Dimensional(3-D) VLSI technology which is considered to be one of the leading areas among the integrated circuit technologies in the next century. Furthermore the researches of 3-D VLSI in Japan are summarized as well.

In the second part of this paper, as a case study, the features of 3-D VLSI are exhibited by testing various layouts of syndrome decoder for double-error correction of linear codes, which can be used for example to improve the reliability of main memories. By the layout results of DEC-BCH decoder, material as well as time savings using 3-D VLSI technology compared to the conventional 2-D VLSI technology are shown.

1 Introduction

Microminiaturization and high density are indispensable to increase the performance of Integrated Circuits (IC) [1][2]. However, various technical problems such as the limitations of chip size and microminiaturization, as well as economical problems come to the fore. In order to alleviate these problems, an attempt has been made to develop Three-Dimensional(3-D) VLSI technology[3][4][5]. The attempt started less than ten years ago[6][7], and now 3-D ICs consisting of up to three active transistor layers have successfully been manufactured. Furthermore, some 3-D VLSI systems have been proposed.

The first part of this paper is a survey of 3-D VLSI technology. Section 2 defines the 3-D VLSI which we will discuss in this paper. In section 3, some representative classification methods for 3-D VLSI are shown. The fabrication process of 3-D VLSI is presented in section 4. In section 5, the benefits of 3-D VLSI are discussed. In section 6, many problems in fabricating 3-D VLSI in addition to problems in designing 3-D VLSI systems are enumerated. In the last section of the first part of this paper, 3-D VLSI research in Japan is summarized.

In the second part of this paper, section 8, as a case study, the features of 3-D VLSI technology[3][4][5] are clarified by testing various layouts of the syndrome decoder[8][9]. Syndrome decoding is a famous brute force attack method to decode any linear code, but in general, it is considered to be impractical due to its high hardware cost. On the other hand, by limiting its usage to Double Error Correction (DEC), which may be used for example to correct errors in main memories[10], the hardware requirements will be drastically reduced, and it is possible to make a single chip DEC-syndrome decoder with the help of VLSI technology. Furthermore, the layout results of the DEC-BCH decoder show material as well as time savings when using 3-D VLSI technology compared to the conventional 2-D VLSI technology.

2 Definition of 3-D VLSI

3-D VLSI which will be discussed in this paper is defined as vertically integrated VLSI chips consisting of more than two active transistor layers.

There are some researchers considering the one-active-layer (OAL) mode, which restricts active devices to the bottom layer, as a kind of 3-D VLSI [3][4][11][12]. They have also proved some of the benefits of this mode. However, the mode is only an extension of the conventional 2-D VLSI because the technology needed to fabricate it requires only multi-level metallization technology. Thus, we exclude the OAL mode from the 3-D VLSI treated in this paper.

3 Classification Methods for 3-D VLSI

3-D VLSI may be classified by several methods. Figure 1 shows the two representative classification methods for 3-D VLSI that are discussed below.

3.1 Usage of each layer

3-D VLSI classified according to the usage of each layer is divided into two classes depending on whether each layer consists of the same or different functions.

3.1.1 Different functions in each layer

This class of 3-D VLSI consists of different functions applied to each layer. 3-D image processors[13], shown for example in Figure 2, and 3-D I/O terminals will constitute the application of this class. 3-D image processors have been recognized to be one of the most suitable application of 3-D VLSI by researchers in Japan.

3.1.2 Same functions in each layer

This class of 3-D VLSI consists of the same functions applied to each layer. 3-D memory, shown in Figure 3, and 3-D systolic arrays, for example, will constitute the applications of this class.

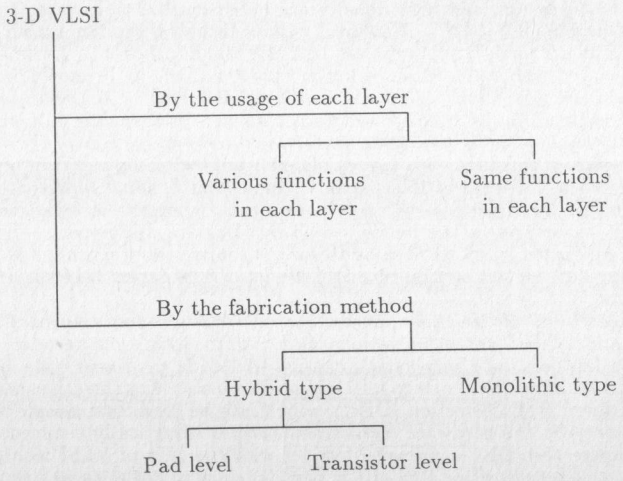


Figure 1 Classification of 3-D VLSI

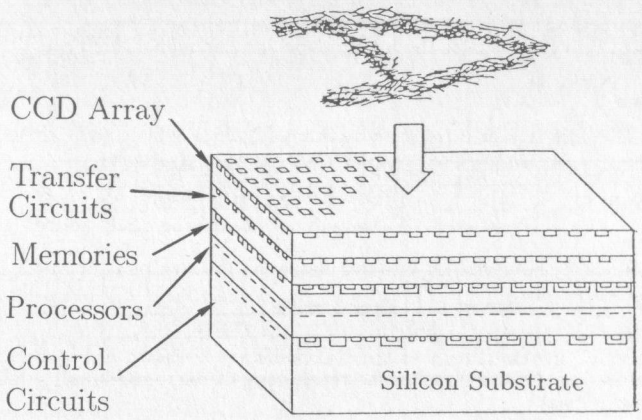


Figure 2 3-D image processor

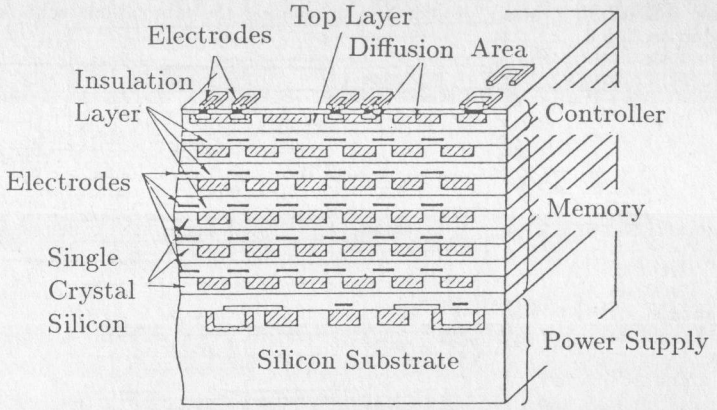


Figure 3 3-D memory

3.2 Fabrication method

There are many methods to fabricate multiple active transistor layers. First, this classification is divided into two classes: the hybrid type and the monolithic type.

3.2.1 Hybrid type

This class of 3-D VLSI is fabricated by attaching IC chips which have already been fabricated. The benefits of this class as compared to the other class are:

1) Easier to fabricate,
2) Higher yield,
3) Fewer technical problems,
4) Many functions can be easily added.

This type of 3-D VLSI can be subclassified into two classes by the level of attachment between ICs.

(a) Pad level

The attachment of ICs in this class is at the I/O pad level as shown in Figure 4. The conception of this class may be the same as the technology of micro-modules. There are some problems with this class:

1) I/O pads are so large compared to transistors or wiring that they prevent high density.

2) The small gaps between the IC chips prevent effective cooling.

(b) Transistor level

The attachment of chips in this class is at the transistor level. ELVIC(Element Level Vertically Integrated Circuit) developed by NEC Corp. in 1984[14], shown in Figure 5, belongs to this class.

3.2.2 Monolithic type

This class of 3-D VLSI, shown in Figure 6, may be the nucleus of 3-D VLSI fabrication in the future. A technology to make the stratum of planar transistors is needed to fabricate monolithic 3-D VLSI. Many problems in developing the fabrication technology will be discussed in section 6.

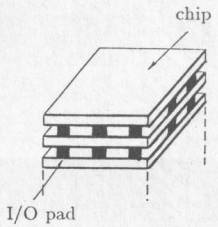


Figure 4 Hybrid 3-D VLSI with pad level

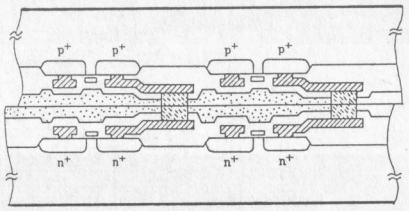


Figure 5 ELVIC

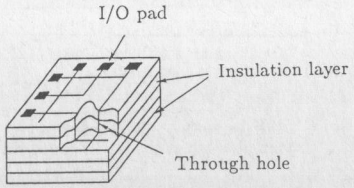


Figure 6 Monolithic 3-D VLSI

4 Fabrication Process of 3-D VLSI

Figure 7 shows a representative wafer fabrication process for 3-D VLSI[5] included in the monolithic type .

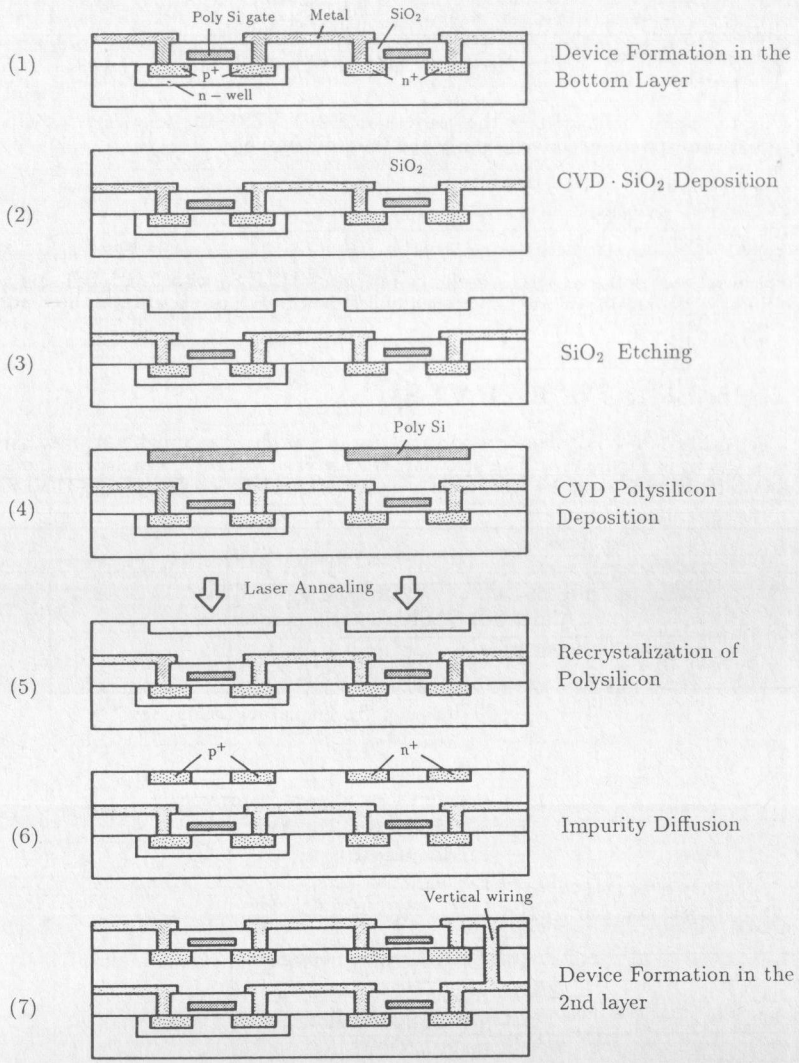


Figure 7 Wafer fabrication process of 3-D VLSI

5 Benefits of 3-D VLSI

In this section, many of the expected benefits of 3-D VLSI are enumerated. First of all, 3-D VLSI is a kind of IC, so that the benefits of 2-D ICs listed below are possessed by 3-D VLSI as well.

1) small size,
2) light weight,
3) mass production,
4) low price,
5) high speed,
6) high reliability,
7) low power dissipation.

Furthermore, compared to the conventional 2-D VLSI, the following benefits may be expected through the effective usage of the third dimension:

1) high density,
2) material savings,
3) high speed.

There have been some research results comparing 2-D VLSI with 3-D VLSI and some case studies have demonstrated the superiority of 3-D VLSI with respect to the three points listed above [3][4][11][15].

6 Problems of 3-D VLSI

To realize 3-D VLSI, there are enormous problems that can roughly divided into the two classes shown in Figure 8. The first class is a set of problems on developing the fabrication process of 3-D VLSI, and the second class is a set of problems on designing 3-D VLSI systems.

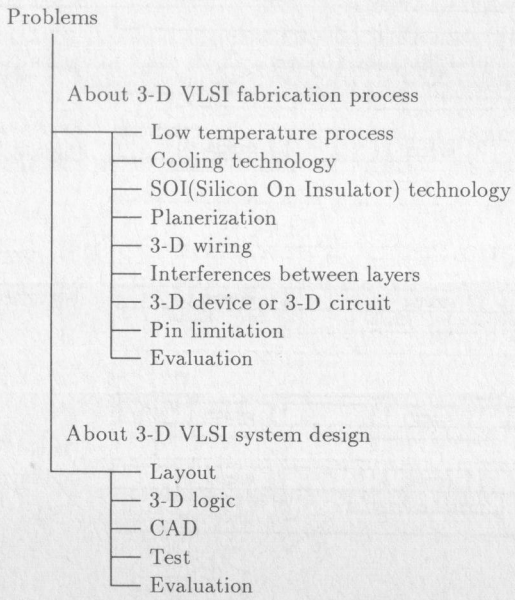


Figure 8 Problems of 3-D VLSI

7 Research on 3-D VLSI in Japan

Research on 3-D VLSI in Japan can be summarized as follows. In 1981, a research and development project on 3-D ICs started to develop advanced device technologies. The Electrotechnical Laboratory and seven companies (NEC Corporation, Toshiba Corporation, Mitsubishi Electric Corporation, Oki Electric Industry Company Limited, Matsushita Electric Industrial Co.,Ltd., Sanyo Denki Co.,Ltd., and Sharp Corporation) are participating in the project. Furthermore, other Japanese companies and also some Japanese universities are participating in the research on 3-D VLSI. The products of 3-D ICs in SSI or MSI forms in Japan are shown in Table 1 [16]~[31] and the proposed systems on 3-D VLSI in Japan is shown in Table 2 [15][32][33][34].

Table 1 The products of 3-D VLSI in Japan

Year	Research group	No. of active layers and their details	Products
1982	Mitsubishi	1 SOI	19-stage ring oscillator
	Toshiba	1 SOI	25-stage ring oscillator
1983	Fujitu	1 SOI	19-stage ring oscillator
		2 SOI -PMOS Bulk-NMOS	7-stage ring oscillator
		2 SOI -PMOS SOI -NMOS	7-stage ring oscillator
	Matsushita	1 SOI	10-bit dynamic shift register
		2 SOI -PMOS Bulk-NMOS	4-bit dynamic shift register
	Mitsubishi	1 SOI	31-stage ring oscillator
		2 SOI -PMOS Bulk-NMOS	45-stage inverter chain
		2 SOI -PMOS Bulk-NMOS	Frequency divider
	NEC	2 SOI -PMOS Bulk-NMOS	19-stage inverter chain
	Oki	1 GaAs/Ge/Si	17-stage ring oscillator
	Sanyo	1 SOI	19-stage ring oscillator
	Sharp	1 SOI	31-stage ring oscillator
		2 SOI -PMOS Bulk-NMOS	Image sensor
1984	Fujitsu	1 SOI	15-stage ring oscillator
		2 SOI -PMOS Bulk-NMOS	7-stage ring oscillator
		2 SOI -CMOS	0.44k gate array
		2 SOI -NMOS Bulk-PMOS	SRAM
		3 SOI -NMOS SOI -NMOS Bulk-PMOS	SRAM
	Hitachi	1 SOI	51-stage ring oscillator
	Matsushita	3 a-Si SOI -NMOS Bulk-CMOS	8-bit linear sensor

	Mitsubishi	2 SOI -CMOS Bulk-NMOS	256-bit SRAM
	NEC	2 SOI -PMOS Bulk-NMOS	53-stage ring oscillator
		2 SOI -PMOS Bulk-NMOS	JK Flip-Flop
		2 SOI -PMOS Bulk-NMOS	32-bit dynamic shift register
	Oki	1 GaAs/Si	17-stage ring oscillator
		1 GaAs/Ge/Si	17-stage ring oscillator
	Sharp	3 a-Si SOI -PMOS Bulk-CMOS	Image sensor
	Toshiba	2 SOI -PMOS Bulk-NMOS	23-stage ring oscillator
		2 SOI -PMOS Bulk-NMOS	4-bit full adder
1985	Mitsubishi	2 SOI -PMOS Bulk-NMOS	10-bit linear sensor
		2 SOI -PMOS Bulk-NMOS	1.1k gate array
	Oki	1 SOI	102-stage ring oscillator
	Sharp	1 SOI	31-stage ring oscillator
1986	Matsushita	2 SOI -CMOS	4k-bit SRAM
	Sharp	3 SOI -NMOS SOI -PMOS Bulk-NMOS	Dynamic linear memory

Table 2 The research on 3-D VLSI systems in Japan

Research group	3-D VLSI systems
ETL	3-D ROM for Kanji Characters
	Retina Simulator
NTT	3-D CAM RAM
Tokyo Institute of Technology	3-D PLA
Keio Univ.	Boolean Matrix Solver
	Polynomial Transformer
	Dynamic Interconnection Networks
	DEC Syndrome Decoder
Shizuoka Univ. and Tohoku Univ.	3-D FFT Processor
Hiroshima Univ.	3-D Shared Memory

8 An example of 3-D VLSI: a Syndrome Decoder for double Error Correction

In this section, as a case study, the features of 3-D VLSI technology, discussed above, are clarified by testing various layouts of the syndrome decoder.

8.1 The syndrome decoding of linear codes

8.1.1 (n, k) linear code and the standard array

The linear code which we consider in this section is a subspace of GF(2^n) and is also called (n, k) linear code [8][9].

The standard array can be tabulated by the following algorithm [9].

1. Let $\vec{0}, \vec{C}_2, \vec{C}_3, ..., \vec{C}_{2^k}$ be the 2^k codewords in an (n, k) linear code.
2. Write all codewords in the first row.
3. Execute the following steps:

   Beginning with the initial value j equal to 1, continue 3.1 and 3.2 until there is no unused word remaining.

   3.1 Choose any previously unused word $\vec{V}_j$ which is as close as possible to the all zero word.

   3.2 Write $\vec{0} + \vec{V}_j, \vec{C}_2 + \vec{V}_j, \vec{C}_3 + \vec{V}_j, ..., \vec{C}_{2^k} + \vec{V}_j$ for the jth row.

Figure 9 shows the standard array. Every row in Figure 9 is called a "Coset" and the words in the first column are colled "Coset leaders".

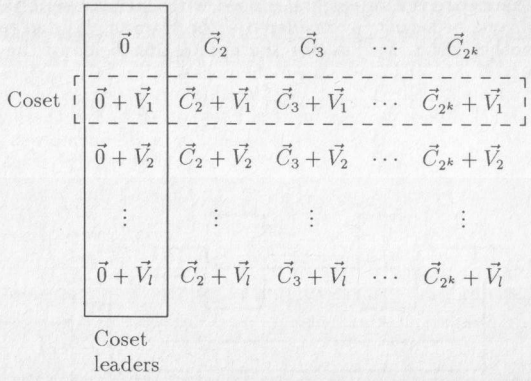

$\vec{0}$ $\vec{C}_2$ $\vec{C}_3$ $\cdots$ $\vec{C}_{2^k}$
Coset
$\vec{0} + \vec{V}_1$ $\vec{C}_2 + \vec{V}_1$ $\vec{C}_3 + \vec{V}_1$ $\cdots$ $\vec{C}_{2^k} + \vec{V}_1$
$\vec{0} + \vec{V}_2$ $\vec{C}_2 + \vec{V}_2$ $\vec{C}_3 + \vec{V}_2$ $\cdots$ $\vec{C}_{2^k} + \vec{V}_2$
$\vdots$ $\vdots$ $\vdots$ $\vdots$
$\vec{0} + \vec{V}_l$ $\vec{C}_2 + \vec{V}_l$ $\vec{C}_3 + \vec{V}_l$ $\cdots$ $\vec{C}_{2^k} + \vec{V}_l$
Coset leaders

Figure 9 The Standard Array

8.1.2 Syndrome decoding

Syndrome decoding is an elementary and famous brute force attacking algorithm using the standard array [8][9]. Let $\vec{V}=[V_i]$ and $H=[h_{ij}]$ be any received word and parity-check matrix, respectively. Then the syndrome $\vec{S}$ can be calculated as follows:

$$\vec{S} = \vec{V} \cdot H^T \tag{8-1-1}$$

where T denotes the transposition of the matrix H.

Every linear code has the property that all vectors in the same coset have the same syndrome unique to that coset. Using this property, it is possible to decode directly from a syndrome which can detect its associated coset leader if the received word has no uncorrectable error.

The algorithm of syndrome decoding is shown as follows:

1 Calculate the syndrome.

2 Look up its associated coset leader.

3 Subtract the coset leader from the received word to correct the error.

Figure 10 shows the basic organization of the syndrome decoder. The Syndrome Generator (SG) calculates the syndrome $\vec{S}=[S_i]$ from the received word following the equation (8-1-2).

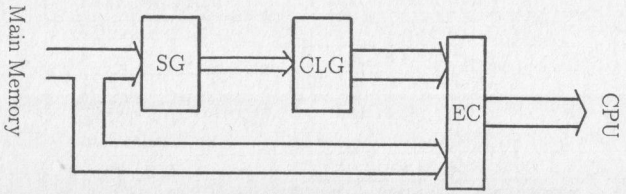
$$S_i = \sum_j h_{ij} V_j \tag{8-1-2}$$

The corresponding hardware is easily designed with exclusive-OR gates. The Coset Leader Generator (CLG) searches for the associated coset leader $CL=[CL_i]$. Thus, CLG is a kind of decoder which can be realized by a pass transistor logic array. The Error Corrector (EC) subtracts the coset leader $CL=[CL_i]$ generated by CLG from the received word $\vec{V}=[V_i]$ following the equation (8-1-3) to generate the corrected codeword $\vec{W}=[W_i]$.

$$\vec{V} = [W_i] = [CL_i - V_i] \tag{8-1-3}$$

The corresponding hardware is also easily designed with exclusive-OR gates.

In general, syndrome decoding is considered to be impractical due to the hardware cost of CLG. On the other hand, by limiting the application fields of the code to such fields as double error correction (DEC) for main memories, the hardware requirements will be drastically reduced, and with the help of VLSI technology it is possible to make a single chip DEC-decoder. Table 3 shows the relationship between n, k, and the number of correctable syndrome patterns (Nc) of the DEC-BCH decoder for main memories. Incidentally, the reason why we have chosen the BCH code is because it requires the least check-bits.



SG : Syndrome Generator
CLG : Coset Leader Generator
EC : Error Corrector

Figure 10 The basic organization of the syndrome decoder

Table 3 The relationship of DEC-BCH code

k	16	32	64	128
n	26	44	78	144
Nc	352	991	3,082	10,441

8.2 VLSI model

A 2-D VLSI model, which can be expanded to the 3-D VLSI models, for our layouts of the DEC-syndrome decoder is shown as follows:

1) The ordinary λ-rule [35] for CMOS design [36] is applied to the layouts of each active level. In this section, we estimate that λ is equal to about $1\mu m$.

2) The size of I/O pads is originally independent of the λ-rule. In this section, we estimate that each I/O pad is $100\mu m$ by $100\mu m$ in size and is separated from adjacent pads by $100\mu m$.

3) Layouts of 3-D VLSI.

   3-1) Stacked CMOS technology [5] is applied so that n-MOS and p-MOS transistors are fabricated on separate layers.

   3-2) The size of through holes is estimated to be the same as contract holes.

   3-3) The number of active layers is limited about 20 because of the limitation of process technology. Thus, any system which requires more than about 20 active layers should be reconstructed to satisfy the limit on the number of active layers.

   3-4) I/O is limited to the top or bottom layers only.

8.3 The outcome of the layouts

Table 4 shows the outcome of the layouts of the DEC-BCH syndrome decoder. Figure 11 shows the block diagram of the 3-D syndrome decoder. And as an example, Figure 12 shows one intermediate layer of the 3-D syndrome decoder with k equal to 128.

Table 4 The outcome of the layouts

Table 4-1 2-D Cases

k	16	32	64	128	Unit
Length	3,258	8,272	19,048	43,200	λ
Width	2,772	4,466	9,506	19,116	λ
Area(A_{2-D})	9,779,616	36,942,752	181,070,288	825,811,200	λ^2
Maximum Wire Length (MWL_{2-D})	3,258	8,272	19,048	43,200	λ

Table 4-2 3-D Cases

k	16	32	64	128	Unit
Length	800	1,000	1,626	2,906	λ
Width	800	1,000	1,640	2,880	λ
Area(A_{3-D})	640,000	1,000,000	2,666,640	8,369,280	λ^2
No. of Active Layers	15	17	19	21	
Volume(V)	9,600,000	17,000,000	50,666,160	175,754,880	λ^2
V/A_{2-D}	98.2	46.0	28.0	21.3	%
Maximum Wire Length(MWL_{3-D})	1,016	1,700	2,990	5,490	λ
MWL_{3-D}/MWL_{2-D}	28.8	20.6	15.7	12.7	%
Remarks	I/O bottleneck	I/O bottleneck			

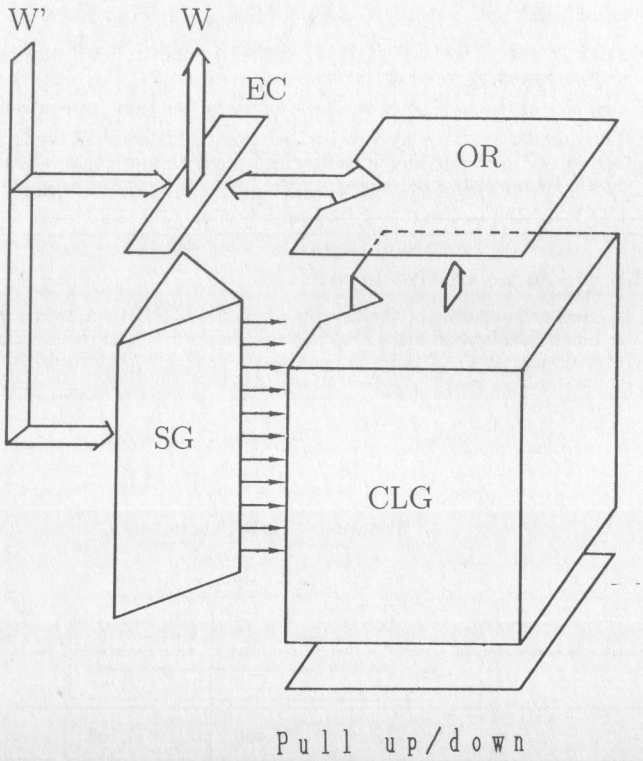


Figure 11 3-D block diagram of the syndrome decorder

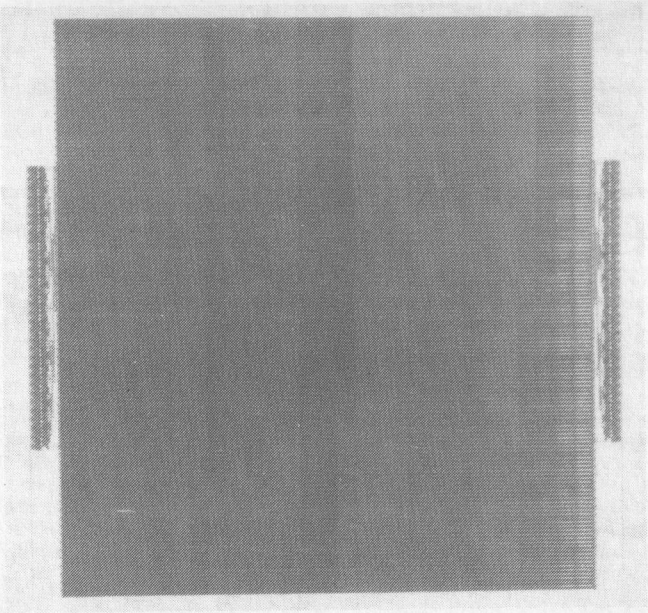


Figure 12 A layout example

8.4 The advantages and disadvantages of 3-D VLSI in the case study

From the outcome of the layouts, the advantages as well as disadvantages of 3-D VLSI are described as follows:

8.4.1 Advantage of 3-D VLSI

1) High density

The reason for this item is derived from two facts: that the wire can be routed effectively using the third dimension and that isolation is not needed when using the stacked CMOS technology. From this item, the material saving by 3-D VLSI compared to 2-D VLSI is shown.

2) High speed

This item is derived directly from the first. From this item, the time saving by 3-D VLSI compared to 2-D VLSI is shown.

8.4.2 Disadvantages of 3-D VLSI

1) I/O problem

Because 3-D VLSI can be more compact than 2-D VLSI, the I/O problem will be more serious. For example, the 3-D syndrome decoder with $k \leq 32$ are typically I/O bottleneck cases. In these cases, the size of the chip is determined from the size of its I/O.

2) Hardware difficulties

There are many hardware difficulties with 3-D VLSI technology at present. For example, 3-D VLSI technology for making more than 20 active layers or to remove heat generated by each layer effectively has not been developed yet. These problems will be resolved gradually by the next century.

9 Conclusion

A survey of 3-D VLSI technology and research on 3-D VLSI in Japan were shown in the first part of this paper. Next, as a case study, the features of 3-D VLSI were clarified by testing various layout results of the DEC-BCH syndrome decoder.

3-D VLSI has many benefits, such as high density or high speed. Since the research on 3-D VLSI technology started less than ten years ago, 3-D ICs manufactured heretofore have been only of the SSI or MSI forms. We hope we will be able to obtain 3-D VLSIs with higher density and higher speed than ever overcoming the many problems which are inherent in 3-D VLSI technology.

We hope that there will be great progress in research to overcome these problems in 3-D VLSI technology.

References

[1] Smith,R.J. : "Electronics: circuits and devices", 2nd ed., John Wiley & Sons, Inc., 1980.

[2] Muroga,S. : "VLSI SYSTEM DESIGN", John Wiley & Sons, Inc., 1982.

[3] Rosenberg,A.L. : "Three-Dimensional Integrated Circuitry", VLSI SYSTEMS AND COMPUTERS, pp.69-80, Computer Science Press, 1981.

[4] Rosenberg,A.L. : "Three-Dimensional VLSI: A Case Study", Journal of ACM, Vol.30, No.3, pp.397-416, July 1983.

[5] Kokubu,A. : "Three-Dimensional Device Project", Tutorial Textbook of the 13th Annu. Int. Symp. on Computer Architecture, pp.75-101, 1986.

[6] Gat,A., Gerzberg,L., Gibbons,J.F., Magee,T.J., Peng,J. and Hong,J.D. : "CW Laser Anneal of Polycrystalline Silicon, Crystalline Structure, Electrical Properties", Appl. Phys. Lett., Vol.41, pp.775-778, 1978.

[7] Gibbons,J.F. : "SOI- a candidate for VLSI?", VLSI Design III, pp.54-55, 1982.

[8] Peterson,W.W. and Weldon,E.J. : "Error Correcting Codes", 2nd Edition, MIT Press, Mass.(1972).

[9] Blahut,R.E. : "Theory and Practice of Error Control Codes", Addison-Wesley Publishing Company, (1983).

[10] Imai,H. and Kamiyanagi,Y. : "A Construction Method for Double-Error-Correcting Codes for Application to Main Memories", (in Japanese), Trans. IECE Japan, Part D, Vol.J60-D, No.10, pp.861-868 (1977).

[11] Preparata,F.P. : "Optimal Three-Dimensional VLSI Layouts", Mathematical Systems Theory, 16, 1-8, pp.1-8, 1983.

[12] Leighton,F.T. and Rosemberg,A.L. : "Automatic Generation of Three-Dimensional Circuit Layouts", proc. of ICCD, pp.633-636, 1983.

[13] Taniguchi,K. : "Three Dimensional IC's and application to high speed Image Processor", proc. of IEEE 7th Symposium on Computer Arithmetic, pp.216-222 (June 1985).

[14] Yasumoto,M., Hayama,H. and Enomoto,T. : "Promissing New Fabrication Process Developed for Stacked LSI's", proc. of IEEE International Electron Devices Meeting, pp.816-819, December 1984.

[15] Kurokawa,T. and Aiso,H. : "Polynomial Transformer", proc. of The IEEE 7th Symposium on Computer Arithmetic, pp.153-158, 1985.

[16] Sakurai,J., Kawamura,S., Mori,H. and Nakano,M. : "Lateral Epitaxial Growth in Poly-Si Film over SiO_2 from Single-Si Seed by Scanning CW Ar Laser Annealing", Japanese Journal of Applied Physics, Vol.20, No.3, pp.L176-L178, March 1981.

[17] Kawamura,S., Sakurai,M., Nakano,M. and Takagi,M. : "Recrystallization of Si on amorphous substrates by doughnut-shaped CW Ar laser beam", Appl. Phys. Lett. 40(5), pp.394-395, March 1982.

[18] Sasaki,N. and Nakano,M. : "A CMOS/SOS Gate Array with a New Customization Technique of Cutting", IEEE Transactions on Electron Devices, Vol.ED-29, No.10, pp.1535-1541, October 1982.

[19] Egami,K., Kimura,M. and Hamaguchi,T. : "Laser Recrystallization of Silicon Stripes in SiO_2 Grooves with A Polycrystalline Silicon Sublayer", Appl. Phys. Lett., Vol.43, No.11, pp.1023-1025, December 1983.

[20] Mukai,M., Sasaki,N., Iwai,T., Kawamura,S. and Nakano,M. : "Single crystalline Si islands on an amorphous insulating layer recrystallized by an indirect laser heating technique for three-dimensional integrated circuits", Appl. Phys. Lett., 44(10), pp994-996, 15 May 1984.

[21] Kawamura,S., Sasaki,M., Nakano,M. and Takagi,M. : "Laser recrystallization of Si over SiO_2 with a heat-sink structure", J. Appl. Phys., 55(6), pp.1607-1609, 15 March 1984.

[22] Aizaki,N. : "Recrystallization of Silicon Film On Insulationg Layers Using A Laser Beam Split by A Birefringent Plate", Appl. Phys. Lett., Vol.44, No.7, pp.686-688, Apr.1984.

[23] Kawamura,S., Sasaki,N., Iwai,T., Mukai,M., Nakano,M. and Takagi,M. : "Electrical Characteristics of Three-Dimensional SOI/CMOS IC's", IEEE Electron Device Letters, Vol.EDL-5, No.5, pp.248-250, July 1984.

[24] Mogami,T., Morimoto,M. and Okabayashi,H. : "Via-Hole Filling with Molybdenum by RF Bias Sputtering", Extended Abstructs of the 16th Conference on Solid State Devices and Materials, pp.43-46, Aug./Sept. 1984.

[25] Nishimura,T., Sugahara,K., Akasaka,Y. and Nakata,H. : "Vertically Integrated MOS Devices with Double Active Layers", Extended Abstructs of the 16th Conference on Solid State Devices and Materials, pp.527-530, Aug./Sept. 1984.

[26] Kawamura,S., Sasaki,N., Iwai,T., Mukai,R., Nakano,M. and Takagi,M. : "3-DIMENSIONAL SOI/CMOS IC's FABRICATED BY BEAM RECRYSTALLIZATION", proc. of The IEEE International Electron Devices Meeting, pp.364-367, December 1984.

[27] Akiyama,S., Yoneda,M., Ogawa,S., Yoshii,N. and Terui,Y. : "Fabrication Technologies for Multilayer CMOS Device", Symposium on VLSI Technology, Digest of Tech. Paper, IV-5, pp.28-29, May 1985.

[28] Nishimura,T., Inoue,Y., Sugahara,K., Nakaya,M., Horiba,Y. and Akasaka,Y. : "A Three Dimensional Static RAM", Symposium on VLSI Technology, Digest of Tech. Paper, IV-6, pp.30-31, May 1985.

[29] Hirose,S., Nishimura,T., Sugahara,K., Kusunoki,S., Akasaka,Y. and Tsubouchi,N. : "A 10-Bit Linear Image Sensor Fabricated in Double Active Layers", Symposium on VLSI Technology, Digest of Tech. Paper, IV-8, pp.34-35, May 1985.

[30] Sugahara,K., Nishimura,T., Akasaka,Y. and Nakata,H. : "SOI/SOI/BULK-Si Triple Level Structure for Three Dimensional Device", Second International Symposium on VLSI Technology, Systems and Applications, Proc. of Technical Papers, 3-2, pp.86-89, May 1985.

[31] Nagai,K., Sekigawa,T. and Hayashi,Y. : "Capacitance Voltage Characteristics of Semiconductor-Insurator-Semiconductor (SIS) Structure", Solid State Electronics, Vol.28, No.8, pp.789-798, Aug. 1985.

[32] Takefuji,Y., Kurokawa,T. and Aiso,H. : "Fast Matrix Solver in GF(2)", proc. of The IEEE 6th Symposium on Computer Arithmetic, pp.138-143, 1983.

[33] Takefuji,Y., Kurokawa,T., Ishizaki,M. and Aiso,H. : "New Mtrix Equation Solvers in GF(2) Employing Cramer with Chio Method", proc. of The IEEE ICPP, pp.47-50, 1983.

[34] Hasegawa,M. and Shigei,Y. : "AT^2=$O(N \log^4 N)$,$T = O(\log N)$ Fast Fourier Transform in A Light Connected 3-Dimensional VLSI", proc. of the 13th Annu. Int. Symp. on Computer Architecture, pp.252-260,1986.

[35] Mead,C. and Conway,L. : "Introduction to VLSI Systems", Addison-Wesley Publishing Company, (1980).

[36] Mavor,J., Jack,M. and Denyer,P. : "Introduction to MOS LSI Design", Addison-Wesley Publishing Company, (1983).

OPTICAL LINK in the DELFT PARALLEL PROCESSOR

an example of MOMI-connection in MIMD-supercomputers

L. Dekker and E.E.E. Frietman, W. Smit, J.C. Zuidervaart

Department of Mathematics & Informatics, Department of Applied Physics
University of Technology
Delft, The Netherlands

1. INTRODUCTION

A blue–print of a parallel (digital) processor with MOMI–connection[1] has already been published in 1975 [Dek1]. A first experimental set–up was constructed in 1976 [Andr et al], consisting of a small number of pocket calculators. In 1981 the prototype DPP81 of the Delft Parallel Processor (DPP) became available [Sips], [Dek2]. The present DPP84 [Brok et al] is a somewhat updated and upgraded version of the DPP81. The performance of the DPP84 will soon be in the range of a minisupercomputer, when each DPP processor is extended with a vector coprocessor. The application in the DPP of optical communication through a holographic backplane was already considered in [Dek4]. Originally the design of an optically accessible memory, based on a space–parallel optic–to–electric transducer, was realized for the detection of eyeball movements [Frie1,2,3] to overcome the disadvantages of CCD camera's, but more important to take advantage of the fact that more degrees of freedom of movement were allowed. The DPP8X under development now is a supercomputer that can offer a 'full interconnectability' MOMI communication system with a speed–up in the range of 10^4 MFLOPS. This MOMI communication system does not rely on carefully terminated coaxial connections, but will be equipped with an Optical Link in combination with a so–called POWERRAM[2] as a MI memory[3]. The DPP8X with its high degree of interconnectability can support the development of novel concepts in computing like for instance the neural network. In such a network each computational unit, repesented by a neuron, must be able to communicate with thousands of others. The realization of this dense connectivity is only feasible with optics. In the DPP8X concept free space optical connectivity shows a promising future. Also important for the future of supercomputation is the fact that supercomputers have to become powerful with respect to parallel symbolic processing. For this reason the processors of the DPP8X will be provided with a POWERCAM[4].

2. COMPUTE- VERSUS TRANSFER-BOUND PARALLEL PROCESSING

In this chapter it will be explained why a communication system of a parallel computer must be of high quality firstly with respect to the interconnection topology and thereafter with respect to the data exchange protocol.

2.1. Interconnection Topology

Consider the problem of solving a task T by means of parallel processing on a MIMD parallel computer with p processors P_i , i=1,...,p , in the case that T has a

---

[1]MOMI = Multiple Output Multiple Input.
[2]POWERRAM: a space–parallel optic–to–electric transducer.
[3]MI–memory = Multiple Input accessible memory.
[4]POWERCAM = electrically content addressable POWERRAM.

parallelism[1] $p_{task} \gg p$. Then parallelization of T will result in a decomposition of T into m parallel subtasks T_i , $i=1,\ldots,m$, where $m=p$. As to the data processing this decomposition often allows a load–balanced implementation of the subtasks T_i , $i=1,\ldots,p$, onto the processors P_i , $i=1,\ldots,p$, of the parallel computer, where P_i takes care for T_i . But even in case of perfect 'data processing' load–balancing the efficiency of the parallel computer can still prove to be low because of inadequate 'data exchange' load–balancing as a consequence of lack of processor interconnectability. At present in practice the processor interconnectability is quite often not sufficient with regard to the interactivity that exists between the subtasks T_i .

In order to facilitate the judging of the quality of an interconnection topology quantitative measures will be introduced for the subtask interactivity and for the processor interconnectability.

Let a task T be decomposed into m subtasks T_i , $i=1,\ldots,m$, where each subtask T_i has a scalar output x_i and a vector input $\mathbf{u}_i$, and let the input $\mathbf{u}_i$ be obtained by selection of $\iota=d_i$ components , $1\le d_i\le m$, from $\mathbf{x}=(x_1\ldots x_m)$ by means of a multiselection operator $\zeta_i=(\zeta_1\ldots\zeta_\iota)$. Figure 2,1 represents the corresponding block diagram.

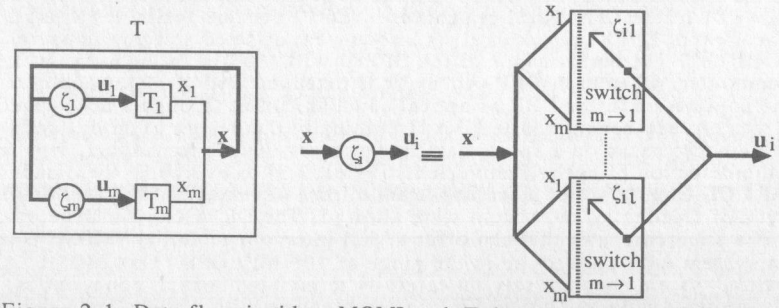


Figure 2,1. Data flow inside a MOMI task T decomposed in m SOMI[2] subtasks T_i .

Whether direct interaction between subtasks exists or not can be described at each data exchange event by means of an interaction matrix, here denoted as data flow matrix F_T with elements f_{ij} , $i,j=1,\ldots,m$. The elements f_{ij} can be either 0 or 1 , where 0 means 'no data flow' and 1 means 'data flow' from subtask T_j to subtask T_i. Here the discussion will be restricted to the case that there are no disjunct subtask clusters.

Note: It can happen that parallelization of a task T into subtasks results in one or more disjunct clusters Tc_i, $i=1,\ldots,j$, of subtasks Tc_{ik}, $k=1,\ldots,\gamma_i$, where 'disjunct' means that as to the cluster interaction these clusters are isolated from each other. An isolated subtask cluster implies that this subtask cluster has neither to send data to other subtask clusters nor to receive data from other subtask clusters. If two subtasks T_i and T_j belong to different isolated subtask clusters then $f_{ij}=0$ and $f_{ji}=0$. For example a cluster of one isolated subtask T_j implies that the j^{th} row and column vectors are zero vectors.

Introduce as a measure for the interactivity ϕ_T of a m*m matrix F_T:

$$\phi_T = {}^{m}\log\left(\frac{\mu}{m}\right) \quad , 0\le\mu\le m^2 \tag{2.1}$$

where μ is the number of nonzeroes in the matrix F_T. The ϕ_T–interval of interest is

[1] parallelism = number of processors that are required if the job is fully parallelized.
[2] SOMI = Single Output Multiple Input.

[-1,1], because $\mu=0$ means no interaction at all.

Notes: 1) The case $\phi_T=1$ corresponds with a full matrix F_T.
2) Examples of the case $\phi_T=0$ are a closed-loop 'pipeline' interaction between all subtasks and a 'single broadcast' interaction[1].
3) If $\phi_T<0$, what corresponds with $\mu<m$, then there are at least two noninteracting (disjunct) clusters of subtasks.

Consider now a parallel computer consisting of p processors with a communication system that allows the realization in one 'data exchange' event of any interaction between b subtasks as can be described in a data flow matrix F_T. An example of such a communication system is one with b busses[2] , where $b \leq p^2$.
Implementation of a data processing job onto a multibuss parallel computer results in a data flow matrix F_p, where element f_{ij} describes the need for data flow from the output of processor P_j to the input of processor P_i . Introduce as a measure for the interconnectability ϕ_p of a p*p matrix F_p:

$$\phi_p = {}^p\log\left(\frac{b}{p}\right) \quad , 1 \leq b \leq p^2 \qquad (2.2)$$

The corresponding ϕ_p–interval is [-1,1], where ϕ_p =-1, 0, 1 means resp.: 1 buss, p busses, p^2 busses.

Note: The case $b=p^2$ is a special case, because then for each processor there are always available p busses. In other words there is full processor interconnectability. However, in stead of using p^2 busses a simpler technical implementation is possible. 'Full interconnectability' can also be realized by means of a communication system composed of a p–tuple broadcast system and p parallel accessible p–tuple input buffer memories, like is applied in the Delft Parallel Processor.

Notice that the above quantative definition of processor interconnectability cannot be applied to parallel computers based on other communication topologies like transputer– and hypercube–type communication systems. The reason is that the number of possible interprocessor connections per data transfer does not uniquely correspond with the number of subtask interactions. In other words then the interconnection topology has much influence on the set of data flow matrices F_T that can be handled in one data transfer per data exchange event. For example, a single broadcast system allows the parallel interaction of 1 subtask to the (p–1) other subtasks, but requires p successive data transfers to realize a pipelined interaction between p subtasks.

For many data processing jobs the task data flow matrix F_T and consequently also ϕ_T will change as a function of time. Figure 2,2 shows a fictitious example of the course of ϕ_T as a function of time. For this example a parallel computer with m subtasks, p=m processors and b=p busses does not provide enough interconnectability to handle the data exchange between m subtasks without applying 'extension in time' of the data exchange [Dek3]. But a parallel computer with p=m processors and $b=p^2$ busses can handle all data exchange events in only one data transfer time τ_d. More extension in time of the data exchange will happen if the subtask interactivity ϕ_T increases compared to the processor interconnectability ϕ_P. Consequently the total data exchange time will increase. In case $\phi_P \ll \phi_T$ there is the risk that the total data exchange time becomes so large compared to the total processing time that the speed–up of a parallel computer becomes transfer–bound

---

[1] 'pipeline' interaction = interaction between subtasks coupled in cascade.
'single broadcast' interaction = one subtask produces data for all processors.

[2] bus = communication channel that can be used by each processor, but one at a time, to send data to one processor, selected out of the other (p–1) processors.

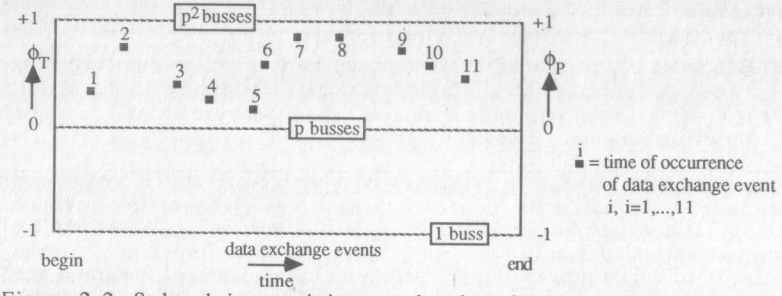


Figure 2,2. Subtask interactivity ϕ_T related to the processor interconnectability ϕ_P.

in stead of compute–bound. In parallel processing the achievable speed–up related to the speed of one processor cannot be more than the number of processors and will often be (much) smaller, because of inefficient use of these processors and insufficient communication capacity. In order to obtain a large speed–up it is necessary to construct compute–bound MIMD parallel computers with many fast processors, but having also a powerful communication system with processor interconnectability ϕ_P close enough to 1. The processor interconnectability ϕ_P must be so close to 1 that only in a very few exceptional cases transfer–bound processing will occur.

It is obvious that a parallel computer with many processors does only make sense in case of large data processing jobs containing much parallelism p_{job}. Then the data processing job must be formulated as a task T, consisting of m parallel subtasks T_i, i=1,...,m, with m=p, where all processors will be active most of the time. For p_{job}»p the subtasks T_i will consist of a sequence of (smaller) subtasks. Notice that in case of ϕ_T=constant, where only for convenience p_{job} is taken equal to p, an increase of p will normally result also in an increase of the processing times τ_{pi} of the subtasks T_i. For a parallel computer consisting of a number of uniprocessors[1] $O(\tau_{pi})$, the asymptotic growth of τ_{pi} for a large number of processors, will be taken in this paper in the range from 'constant' up to 'proportional to p'. An example of $O(\tau_{pi})$=const. is the interaction described by a band–matrix, like happens in 'distributed parameter system' solving; an example of $O(\tau_{pi})$=p is a 'numeric linear algebra' problem: a full matrix times a vector.

Notes: 1) If the p subtasks T_i, i=1,...,p, would be implemented on a uniprocessor, then instead of the interprocessor data transfers there arise extra CPU/memory data transfers. In this paper for convenience it is assumed that a CPU/memory data transfer takes the same time as an interprocessor data transfer. For unequal data transfer times the formula (2.3) etc. change, but the conclusions as to 'transfer–bound' and 'compute–bound' remain qualitatively the same. The results for unequal data transfer times will be published later.

2) In this paper it is assumed that $\tau_d \ll \tau_{pi}$, i=1,...,p.

Let p=m=2^q and b=2^r, q,r=integer. In case of b busses μ interactions, μ=1,...,b, can be handled in one data transfer: n_d=1, where n_d represents the number of data transfers that is needed to execute a data exchange event. In general n_d=ν for μ=(ν–1)b+1,...,νb, ν=1,...,b or in other words: $n_d=\lceil(\mu/b)\rceil$, where $\lceil..\rceil$ denote the 'entier ceiling function'.
Figure 2,3 demonstrates for an example with m=p=16 the influence of the number of busses b on the data transfer by giving the course of the total transfer time

[1]uniprocessor = digital processor where all data processing is done sequentially.

$(n_d\tau_d)$ per data exchange event as well as of the running sum $(n_d\tau_d)$ as a function of the running sum of τ_{pm}. The time series of ϕ_T and of τ_{pm} have been taken resp. as follows:

ϕ_T= -0.25, +0.62, -0.32, -0.58, -0.28, +0.82, +0.75, +0.86, +0.65, -0.24
τ_{pm}= 7, 13, 7, 3, 10, 14, 10, 17, 11, 8

where ϕ_T fluctuates from loosely coupled to tightly coupled.

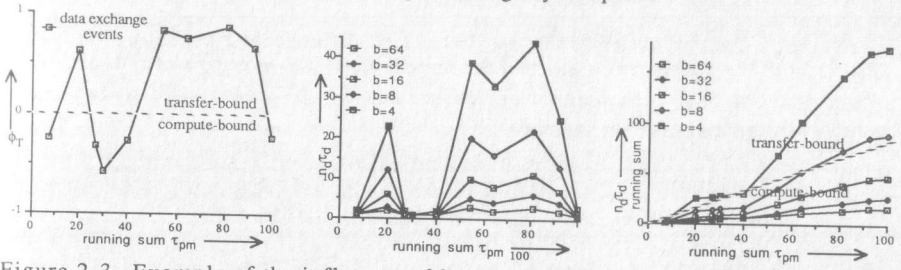


Figure 2,3. Example of the influence of b on n_d as a function of the running sum of τ_{pm} for p=m=16 and b=4,8,16,32,64

It is obvious that for large p the realization of 'full interconnectability' is technically a problem not easy to solve. For instance in practice 'full interconnectablity' by means of p^2 separate busses will technically only be possible for small p for several reasons to be explained later on in section 2. Moreover for large p a 'full interconnectability' communication system will become expensive, because for large p its 'hardware complexity' is proportional to p^2. At present a main question is if there is a technical solution that allows the construction of full interconnectablity at a reasonable price for let us say 1000 processors. Because in parallel processing for a good performance 'data processing' and 'data flow' are of equal importance, we think that a reasonable price for the communication system of a MIMD parallel computer is in the range of half the price of the total system.
As to the practical need of a 'full interconnectability' system it is important to be aware that for small m the value of ϕ_T can often be close to 0 or close to 1, while for large m it will not often happen that ϕ_T is close to 1.
Therefore the problem of 'compute–bound versus transfer–bound' will be considered for relevant combinations of the number of processors p , the number of busses b, the subtask interactivity ϕ_T and the processing time τ_{pi} of subtask T_i. These combinations are taken from:

- small p; large p,
- b=constant; b=p; $b=p^2$
- $\phi_T \approx 1$ (tightly–coupled subtasks); $\phi_T \approx 0$ (loosely–coupled subtasks),
- $O(\tau_{pi})$=constant; $O(\tau_{pi}) \div 1+q\log p$, q=2,10 ; $O(\tau_{pi}) \div p$,

As a global measure for loosely–coupled and tightly–coupled the values of ϕ_T is taken 0 and 1, resp. Then the formula for the speed–up s_{tight} and s_{loose} in the cases tightly–coupled and loosely–coupled are, resp.:

$$s_{tight} = \left(\frac{p\tau_d+\tau_{pa}}{\frac{p^2\tau_d}{b}+\tau_{pm}}\right) p \qquad (2.3a)$$

$$s_{loose} = \left(\frac{\tau_d+\tau_{pa}}{\frac{p\tau_d}{b}+\tau_{pm}}\right) p \qquad (2.3b)$$

In these formula τ_{pa} and τ_{pm} mean the average and the maximum of τ_{pi} , resp.. For

small p it holds:

$$s_{tight} \approx p \ , \ \frac{p^2 \tau_d}{b} \ll \tau_p \ , \ \frac{p\tau_d}{b} \ll \tau_p \quad , \ i=1,\ldots,p \tag{2.4a}$$

$$s_{loose} \approx p \ , \ \frac{p\,\tau_d}{b} \ll \tau_{pi} \quad , \ i=1,\ldots,p \tag{2.4b}$$

For large p the parameters b and τ_{pi}, i=1,...,p, influence the speed-up. In table (2,1) both the speed-up s_{tight} and the speed-up s_{loose} are given for typical combinations of b , the number of busses, and $O(\tau_{pi})$, the asymptotic behaviour of the ratio $\{\tau_{pi}/(\tau_{pi})_{p=1}\}$ for large p:

-	b = constant,	b = p,	$b = p^2$	(2,5)
-	$O(\tau_{pi})$ = constant=1,	$O(\tau_{pi}) \div 1+2\log p$,	$O(\tau_{pi}) \div p$	

Table 2,1. Speed-up for large p.

	b = const.	b = p	$b = p^2$
$O(\tau_{pi}) = 1$	$s_{tight} \approx b$	$s_{tight} \approx p$	$s_{tight} \approx \rho_2^{-1} p^2$
$O(\tau_{pi}) \div 1+2\log p$	$\approx b$	$\approx p$	$\approx \rho_2^{-1} (2\log p)^{-1} p^2$
$O(\tau_{pi}) \div p$	$\approx \rho_1 b$	$\approx p$	$\approx p$
$O(\tau_{pi}) = 1$	$s_{loose} \approx \rho_1 b$	$s_{loose} \approx p$	$s_{loose} \approx p$
$O(\tau_{pi}) \div 1+2\log p$	$\approx \rho_1 2\log p\, b$	$\approx p$	$\approx p$
$O(\tau_{pi}) \div p$	$\approx p$	$\approx p$	$\approx p$

$\rho_1=(\tau_{pa}/\tau_d)_{p=1}$; $\rho_2=(\tau_{pa}/\tau_d)_{p=1}$

The behaviour of the speed-up s as a function of p in dependence on typical combinations of b, ρ, O will now be summarized in some figures, where both p and s are given on logarithmic scales. Let the asymptotic behaviour of the achievable speed-up s for large p be denoted as s_∞. For convenience it is assumed that $\tau_{pi} \approx equal = \tau_p$, i=1,...,p , what is equivalent with the assumption that the subtask load-balancing has been successful. Thus $\rho_1=\rho_2=\rho$, what results in the following simplified formula for s_{tight} and s_{loose} :

$$s_{tight} = \left\{ \frac{p+\rho O(p)}{\frac{p^2}{b}+\rho O(p)} \right\} p \tag{2.6a}$$

$$s_{loose} = \left\{ \frac{1+\rho O(p)}{\frac{p}{b}+\rho O(p)} \right\} p \tag{2,6b}$$

Figure 2,4 gives for s_{tight} an impression of the influence of ρ on the speed-up.

Notes: 1) For b=constant the speed-up s_{tight} has a maximum $s=\sigma_b$ for $p=p_b$ with $p_b>b$. For large b it holds $p_b \approx 2b$ and $\sigma_b \approx b$.

2) For larger values of ρ the curves b=1, $b=p^2$ remain for large p longer close to the curve b=p, while for small b and p the speed-up is larger.

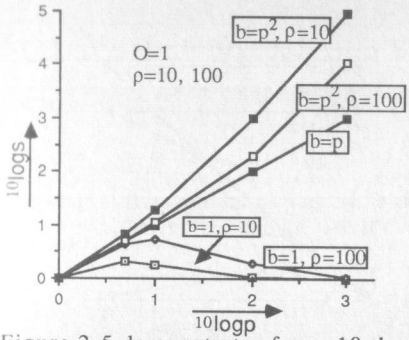


Figure 2,4. Speed-up s_{tight} as a function of p in dependence on b and ρ for $O=O(\tau_p)$

Figure 2,5 demonstrates for ρ=10 the influence of the asymptotic behaviour of $O(\tau_p)$ for large p on s_{tight} and s_{loose} for b=1 and $b=p^2$.

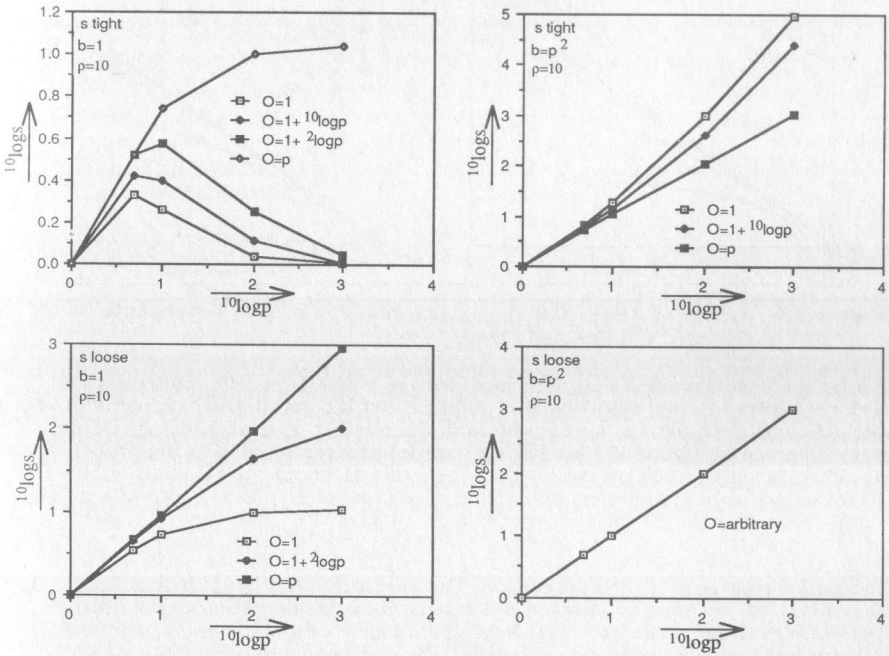


Figure 2,5. Speed–up as a function of p in dependence on $O(\tau_p)$ for b=1 and $b=p^2$.

In case of 1 buss s_∞ is very limited both for tightly coupled and loosely coupled subtasks: $s_\infty=1$ and $s_\infty=\rho(=10)$, respectively. Only for $O(\tau_p)\approx p$ and loosely coupled subtasks s_∞ is proportional to p. Notice that this situation is typical for types of applications, where distributed (parallel) processing can be applied successfully. For $b=p^2$ and loosely coupled subtasks s_∞ is proportional to p. In case of tightly coupled subtasks $s_\infty=p$ for $O(\tau_p)\approx 1$; in the other two cases s_∞ grows faster than p. For $b=p^2$ it even holds: $s_\infty=p^2$.

Figure 2,6 represents s_{tight} and s_{loose} as a function of p with b as parameter in two cases O=1 and $O=1+^2\log p$. For tightly coupled subtasks the speed–up becomes fast saturated after p=b . For loosely coupled tasks the growth of the

speed–up slows down more gradually.

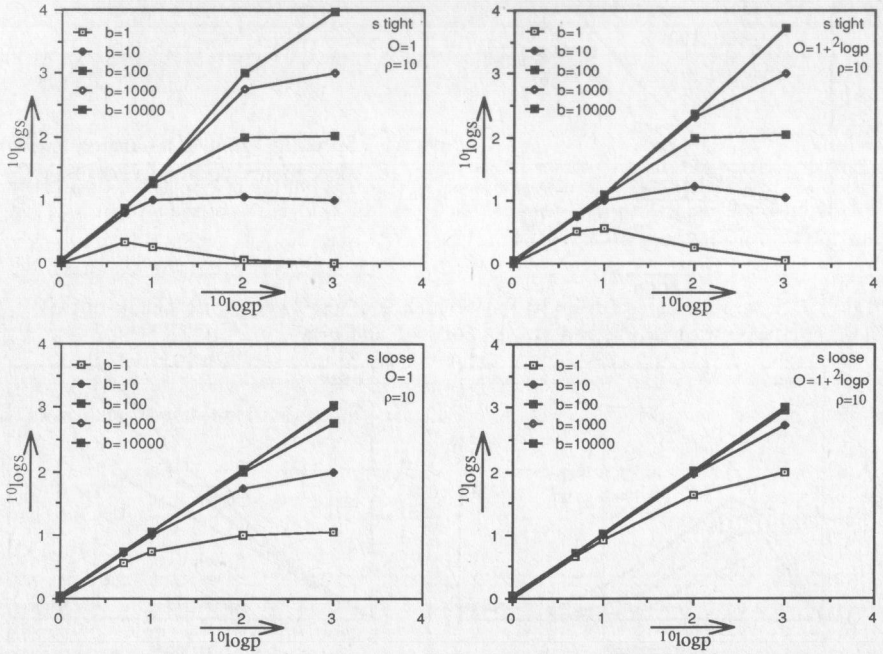


Figure 2,6. Speed–up s_{tight} and s_{loose} as a function of p in dependence on b for $O(\tau_p)=1$ and $O(\tau_p)=1+^2\log p$.

So far we have assumed that the processors in a MIMD parallel computer are uniprocessors. Using uniprocessors implies that the parallelism available in such a MIMD parallel computer is not more than the number of processors p . More speed–up can be obtained by utilizing parallel processing also at the processor level. Then factually each processor of a (general–purpose) MIMD parallel computer becomes a parallel computer, consisting of a number of processing units q. The parallelism of a MIMD parallel computer is increased in this way by a factor equal to this number of processing units [Dek4]. The parallelism still left in a processor subtask can be utilized to decompose it in smaller subtasks for implementation onto the processing units of the processor. The execution of a processor subtask takes less time, but it requires interaction between its subtasks. Hence, the creation of parallelism at the processor level implies that each processor must be provided with a 'processor' communication system for the interconnection between its processing units. Notice that a decrease of τ_p diminishes the value of ρ at the processor level.

For numeric applications parallelization at the processor level by means of parallel/sequential algorithms[1] often results in $qO_p(\tau_p) \approx (1+^2\log q)O_s(\tau_p)$, where $O_p(\tau_p)$, $O_s(\tau_p)$ are the asymptotic growths of the subtask processing times for large p in a MIMD parallel computer and in a uniprocessor, respectively [Dek4], [Dek5], [Dek6]. For convenience we assume here that 'full interconnectability' processor communication systems are applied. Then substitution in (2.6) gives:

[1]parallel/sequential algorithm = incompletely parallelized algorithm with a reasonable speed–up and an efficiency close to 1.

$$s_{tight} = \{\frac{p+\rho O_s(p)}{\frac{p^2}{b}+\frac{\rho(1+^2\log q)O_s(p)}{q}}\}\,p \qquad s_{loose} = \{\frac{1+\rho O_s(p)}{\frac{p}{b}+\frac{\rho(1+^2\log q)O_s(p)}{q}}\}\,p \qquad (2.7)$$

Consider now the case that s_{tight} and s_{loose} are maximized with respect to b, (corresponding with $b=p^2$ and $b=p$, resp.) and that the parallelism q at the processor level is taken equal to p. In this case table 2,2 demonstrates for large p the increase of the speed–up as a result of parallelization at the processor level.

Table 2,2. Increase of speed–up for large p by 2-level parallelization.

		b = p	b = p^2
$O_S=O_p$	$O_S=1$	$s_{loose} \approx p$	$s_{tight} \approx \rho^{-1}p^2$
	$O_S=(1+^2\log p)$	$\approx p$	$\approx \rho^{-1}\,(^2\log p)^{-1}p^2$
	$O_S=p$	$\approx p$	$\approx p$
$(1+^2\log q)O_S=qO_p$	$O_S=1$	$s_{loose} \approx \rho p$	$s_{tight} \approx p^2$
$q=p$	$O_S=(1+^2\log p)$	$\approx \rho(1+^2\log p)p$	$\approx p^2$
	$O_S=p$	$\approx (1+^2\log p)^{-1}p^2$	$\approx (1+^2\log p)^{-1}p^2$

Also in symbolic processing a considerable gain of the speed–up can be obtained by multilevel parallelization.

The processing devices of a parallel computer are physically separated. So there has never been a fundamental objection to apply multiinput and multioutput access to clusters of processing devices. At present it is technically possible to realize parallel interaction between many processing devices by means of a MOMI communication system. An electro–optical approach is described in chapter 3 of this paper.
The situation is quite different for the interaction between a number of processing devices and one common memory. In the recent past the input and output access to memories was based on a so–called 'cross–bar switch' concept; as a consequence at that time memories can be considered to be SISO devices. This fact already made it difficult to apply successfully a 'data processing protocol', where an important part of the interprocessor interaction has to be realized through the medium of one shared memory. Then 'write/read memory access' conflicts will frequently occur during a parallel run. Hence, for more tightly coupled subtasks it will be difficult to avoid transfer–bound parallel processing. This way of transfer–bound processing will not happen if the 'data flow' architecture is based on the ultimate use of local memories.

Notes: 1) The Delft Parallel Processor has a 'full interconnectability' communication system and is based on the ultimate use of local memories during a parallel run. But applying the 'newspaper' method (in the case of one common newspaper for all processors) one can say that the newspaper is a small note–book common memory. The content of this small virtual common memory is duplicated in parallel in the M(ultiple) I(nput) memories of all processors The content of the virtual memory is refreshed each data exchange event.
2) Recently more and more configurations of a memory structure show up, in which the concept of multiread and multiwrite ports are combined. Simultaneous accesses in a single cycle are possible, where each access can either be a read or write action allowing concurrent write/read operations. Multiport memories are still based on a multi 'cross–bar switch' structure. Hence, these memories cannot provide a breakthrough as to the memory access conflicts.

But there is quite an other chance to get transfer–bound parallel processing because of 'write/read memory access' conflicts. That will happen if parallel processing is used as a fast way to reduce enormous amounts of data. Then a parallel computer must be capable to consume these input data in the same speed from the environment of the parallel computer. For example in case of 'on–line' parallel signal processing it raises the problem that the input data must be distributed in parallel over the processors. In case that the input data has to come from a large background memory there is one more problem to be solved: fast 'read' access to the background memory. The only practical way to solve this problem is the realization both hardware– and software–wise of large physically distributed memories. In such a memory device it must be possible to read and write many data in parallel in its submemories and also to distribute many data in parallel to many destinations. Much paraliel data processing of the type 'selection and routing of data packages' is required. In this way a background memory will become a 'parallel data supplier'.

Notes: 1) In the Delft Parallel Processor parallel signal processing can easily be handled by distributing the input data from the environment through the MOMI communication system over some extra parallel channels to the processors. In this way the newspaper is enlarged by the 'news' coming from the environment.
2) The DPP8X will be provided with a 'parallel data supplier'.

An important conclusion is that successful parallel processing demands a.o. much processor interconnectability. Adequate interconnectability is one of the major problems that has to be better theoretically founded. Much innovative work has still to be done.

2.2. Data Exchange Protocol

It will now be explained why a communication system of a MIMD parallel computer must also be of high quality with respect to the data exchange protocol. For convenience firstly it is assumed that there are p processors and that there is one task T decomposed in m=p interacting subtasks. Subtask interaction means that output variables of subtasks are input variables of one or more other subtasks. Subtask interaction has to be realized in a parallel computer by means of data exchanges between processors, initiated at the occurrence of 'data exchange' events. There exists a logic ordering of the 'data exchange' events according to the interaction matrix between the subtasks in the course of time. This logic ordering must be preserved during the execution of a parallel run. The type of both the data exchange protocol and the interconnection topology has much influence on the hardware/software implementation of the execution of the data exchange events.

There exist several conceptual frameworks for information exchange applied in a parallel computer like:
- 'sender to addressee(s)' concept,
- 'newspaper to subscribers' concept,
- 'consumer to resource(s)' concept

The 'consumer to resource' concept is applied for communication between processors and a shared memory, but normally not for interprocessor communication. Let us consider the 'sender to addressee(s)' framework in the case that the way of programming of a parallel computer allows that each processor can autonomously 'recognize' when intermediate data has to be sent to one or more other processors. In order to realize that each processor can become autonomously a 'sender' processor it is necessary that:
- a processor is capable:
  - to process 'information exchange' condition(s) in order to determine the times, at which for this processor there arises the need for a 'data exchange' event: either a 'receive' event to receive data from its environment or a 'send' event to send data to its environment,
  - to edit a data package, consisting of its 'name', the 'message', 'names of

addressee processors' and the 'identification label' corresponding to each of its 'send' events,
- to identify its 'receive' events.

- the need for a 'send' event is reported by a 'sender' processor to the communication system by sending a request to effectuate this 'send' event.
- after a request the communication system gives the 'sender' processor as soon as possible the opportunity to transfer the 'message' data package to the 'addressee' processors through the medium of the communication system. If at a request the communication system is still occupied in transferring a 'message' data package of a predecessor 'sender' processor, then the successor 'sender' processor has to wait until this transfer is ready.

The preservation of the logic ordering of the data exchange events requires that the logic state of the ongoing processing task is an input data to the condition(s) for sending the corresponding message(s). Consequently in order to be capable to process its condition(s) for sending message(s) it is necessary that:
- each processor is continuously informed about the logic state of the ongoing processing task.

In daily practice communication systems based on the 'sender to addressee(s)' framework and the 'consumer to resource(s)' framework have originated from applications where:
- almost never many 'senders' or 'consumers' appear at the same time,
- the data transfer delay time was not very critical, allowing much 'overhead' in the implementation of a data exchange protocol.

In parallel computers in case of 'loosely coupled' and not too small subtasks both data exchange protocols can be realized by means of a communication system that is based on one data buss: SOSI data transfer. For 'tightly coupled' and small subtasks fast MOMI data transfer is required in order to avoid transfer–bound processing. But that will be difficult to realize cost–effectively for these data exchange protocols. Also the preservation of the logic ordering of the data exchange events will introduce a considerable programming overhead.

The 'newspaper to subscribers' framework or briefly the 'newspaper' method is based on a way of information exchange as is applied by publishers of journals to distribute in parallel 'news' to all subscribers. Subscribers of a journal will normally not consume all information sent to them. Each subscriber selects from the 'news' each time a newspaper is offered to him that part of the 'news' what is then of importance for him. Conceptually the hardware/software architecture of a newspaper–based communication system will be roughly as follows.
First let us consider a small parallel computer with a small number of processors. Then it is obvious that all processors are 'subscribers' of one and the same newspaper. Each processor sends all news of importance for other processors to the communication system as a contribution to the actual edition of the newspaper. (In the DPP84 the default value of the news contribution per processor is one data word). A processor has to stop its data processing when information is needed from the next newspaper issue and has to wait until this issue is distributed. The ordering of the interactions between subtasks are ordered according to a logic structure. This logic structure has to be preserved in the course of the subtask processing. For that purpose in a subtask program 'pause' instructions can be used based on some simple criterion.
The communication system sends the successive newspaper editions to all processors. Each processor selects out of each edition the information needed to proceed its data processing subtask.
For a parallel computer with many processors the use of one common newspaper for all processors is not an adequate solution. In case of one common newspaper normally the processors need only very small individual parts of the total news. Using one newspaper also implies that when a processor subtask needs data from the interprocessor environment further processing of this subtask has to wait on the next issue of the common newspaper is ready for distribution. But that means that frequently the processing of subtasks is interrupted without a strict need. These unnecessary waiting times would result in a decrease of the efficiency and as

a consequence also of the speed–up. For large parallel computers one could try to apply a multinewspaper method based on a hierarchical clustering of processors in two or more levels. Then at each level inside a cluster the data exchange protocol is based on the use of a local 'cluster' newspaper, where a part of this newspaper is reserved for news from the cluster environment. From each 'cluster' news there will normally be selected as a contribution to the next issue of the 'higher level cluster', where it belongs to. But, for a $\phi T(t)$ fluctuating significantly in the course of a parallel run the above static clustering of the processors will not be an adequate solution. However, based on the logic ordering of the data exchange events a one–level dynamic clustering of the processors is possible. The DPP8X will be provided with dynamic clustering; the applied method will be published later. Although full interconnectability is strictly only needed within each cluster, dynamic clustering of processors still requires a 'full interconnectability' communication system for the total parallel computer, because the cluster topology will not be constant during a parallel run.

3. THE ELECTRO-OPTIC COMMUNICATION SYSTEM.

3.1 Conceptual Aspects of the EOCS.

The communication architecture of the Delft Parallel Processor is based on local selection of the required data items out of an enormous amount of data, received from other PE's as 'newspapers' during a parallel run. Hence, next-generation DPP's can make use of the most important advantage of optics: its ability to create dense parallel interconnection networks, while "simple " electro-optic and opto-electric transducer circuits can be applied for sending and receiving the data. For the transmission of optical energy between one transmitter and several receivers both free space optics and optical wave guides may be used. In either case implementation of the interconnection network takes advantage of the fact that photons, unlike electrons, do not interact . In particular applying free space optics, like lenses, prisms and mirrors also enable the transfer of billions of space-resolvable points in parallel.

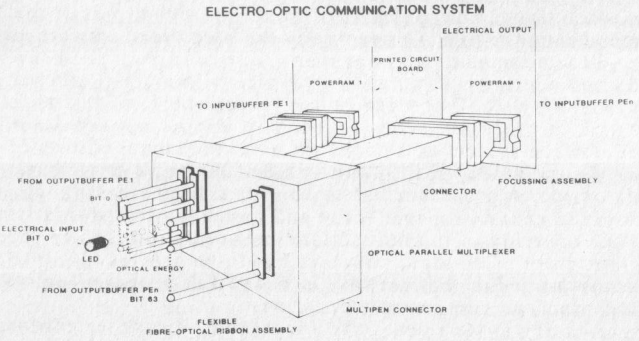


Figure 3,1. EOCS: Electro–optic communication system .

The communication system is illustrated in figure 3,1. The electrical results from the output of each Processing Element are converted by Light Emitting Diodes (LED's) into optical signals. The energy of each LED has to be distributed either over a bundle of "p" multimode fibers ("p" = number of computing elements; in the case of the DPP8X at least 1000) or over an image consisting of "p" optical par-

allel data channels. It has to be realized, however, that the implementation of a full parallel multibroadcast interconnection concept between a thousand PE's requires the routing of "p*p*N" fibers (N = 64 bit wide computer word in the case of the DPP8X). Using merely fibers for this routing, only a restricted number of channels can be applied due to the fact that a great number of interconnections also needs the availability of special components, like for instance special packages together with a very precise alignment of the fiber focussing assembly at receiver side. Notice that demountable multipin connectors, containing the amount of fibers, as required in the DPP8X communication architecture, are for instance still not available.
The most promising solution for the near future is the implementation of an Optical Parallel Multiplexer. In this concept two alternatives are to be considered. In the 'multibroadcast' concept the distribution of the data from one source to "p" ("p" = number of computing elements) channels can be realized with "p" optical starcouplers. In the case, where each PE produces an N bit long computer word, the number of applied starcouplers will be "pN". The second alternative is based on composing and projecting of the complete 'single-broadcast' image, which then contains "pN" pixels.
A fiber optic transmission system can be much smaller in size, lower in costs and lighter in weight than an electrical equivalent of the same bandwidth. Even with the required amount of fibers the dimensions of such an interconnection system are still manageable. Distances to be bridged in the DPP8X will not exceed 5 meters. The availability of Computer Aided Design / Computer Aided Manufacturing (CAD/CAM) techniques allow the design and realization of this rather ambitious goal of mutually connecting a thousand PE's by means of optical fibers in this EOCS.

Even more important than the implementation of optic data channels is the design of components, such as transmitters for converting the electrical signals into light and receivers for performing the reverse conversion. Not only will the cost of these components contribute significantly to the total cost of the PE, but the use of the current technology might also potentially be the bottleneck in the EOCS. In this respect an important problem to be solved is the simultaneous conversion and parallel storage of the optical data. To realize a high-bandwidth digtital signal link a special hardware component at the receiver side has been developed; a small prototype is available. The design of such a component relies mainly on the requirements of parallel loading and storage of 64,000 bits of information, in the case of 1000 PE's and the use of a computer word with a length of 64 bit.

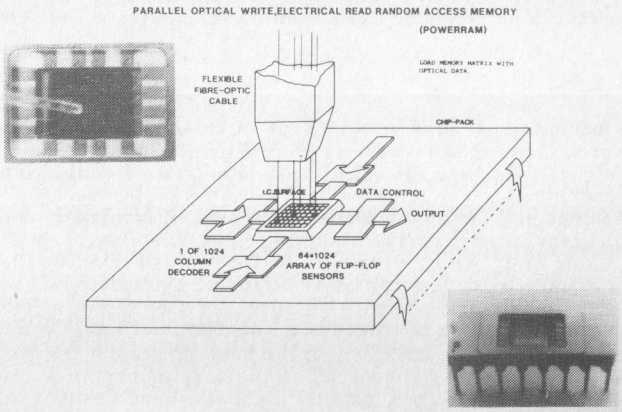


Figure 3,2. POWERRAM: Parallel optical write, electrical read random access memory

A proposal has been formulated for the development of a bipolar 'Parallel Optical Write sequential Electrical Read Random Access Memory' (POWERRAM,TM; Dutch patent 86.02684) based on an array of 1000*64 optical written flip-flop elements; see figure 3,2. The function of this POWERRAM is to convert in parallel the space-parallel presented information, that is conducted through the 1000*64 multimode fibers into a collection of electrical signals with a resolution of 64 bits.

3.2 Design Aspects of the EOCS.

Briefly the requirements will now be specified for a medium speed EOCS prototype implemented with optical fibers. Technical aspects with more details can be found elsewhere [Frie4].
The transmitter side is realized using a specially designed array of high luminescent, low costs LED's. The rise time of the chosen LED will be in the order of 5 ns, where 10 μW of optical power is available to be coupled into each fiber. The fiber to be used in the back plane is a polymer fiber with a core , which will be be in the range of 200 μcrons. The application of polymer instead of glass fibers introduces the disadvantage of higher losses, what is, however, acceptable in an EOCS with short links of maximally 5 meters. The reason for this is that polymer fibers are easy to handle and less costly: properties of extreme importance in a large-scale EOCS.
The quality of the complete transmission system, including the electro-optic and opto-electric transducers, is expressed in the probability of the occurrence of errors. The uncertainties, due to the presence of noise, as to the fluctuations in the pulse duration and amplitude usually are the causes of the occurrence of errors. Hence the amount of optical power reaching the optical detector determines the so-called Bit Error Rate (BER). When ideal conditions are assumed for the detector as well as for the transmitted binary signals, the required BER of the complete EOCS is estimated to be one bad bit out of 10^9 bits.
Measurements were performed on the EOCS with 16 digital interconnections, 5 meters of polymer fiber, permanently glued to the LED on one side and to the POWERRAM on the other side.The connection was made by first precisely positioning and then carefully fastening the fibers to the chip. A perspex coupler was developed in order to make a permanent and more efficient coupling possible. The overall data bit rate proved to be 20 Mbit/s per fiber connection. When these results are extrapolated to the future POWERRAM, equipped with an array of 64*1000 storage elements, the total data transfer bit rate will be in the range of 10^6 Mbit/s. For a comparison, this equals the transmission of a text of about half a million paperbacks per second, while sending four pages at the same time every 70 ns.

3.3 Future Trends.

The future technological edge goes into the direction of the application of ultrahigh-speed GaAs digital circuitry. The main advantages of the GaAs technology are the higher speed and the ability to combine fast digital circuitry with optical and linear components. Research in the field of opto-electronic devices, such as LED's, LASER's and PIN-photodiodes composed of compound GaAs (III-IV) combinations is still in progress. Awaiting the availability of such sophisticated components, the integration of these opto-electronic devices together with high-density Si circuitry by depositing GaAs on top of a processed Si wafer is a very promising step in between.
The addition of much more intelligence to the design of the POWERRAM, in order to increase the amount of parallelism in the next-generation DMIMD[1] super-computer (the DPP8X), has already led to the development of a new opto-electronic device, a Parallel Optical Write, parallel Electrical Read Content Addressable Memory: the POWERCAM (TM; European patent pending).
The necessity to connect a large number of pins of the nowadays present VLSI and

[1]DMIMD = Distributed MIMD

VHSIC components in an electrical way is urgent. So optical interconnections are also being considered at the chip level, because of the many advantageous features over wire connections. The present alternatives are the application of discrete opto-electronic components or a hybrid or monolithic integration. Optical intra chip links are becoming more and more attractive when chips are made as a combination of GaAs on Si.

Acknowledgements.

The ongoing research and development as to the realization of the DPP8X has been stimulated by an INSP grant from the Dutch Government. This grant has a.o. made it possible to establish a cooperation with W. Smit of the ENKA/AKZO as to the realization of the EOCS. The authors would also like to thank E.H.Nordholt and Chr. van Maaren from the Faculty of Electrical Engineering of the Delft University of Technology for their advices and helpful discussions regarding the design of the basic opto-electronic logic cell configuration. We would also like to thank P. van Zuylen and Th.J.M. Jongeling from the TNO Institute of Applied Physics (TPD) for attributing a number of helpful contributions dealing with free space media in the final stages of formulating our research on optical interconnections within our next generation MIMD supercomputer: the DPP8X.

References

[Andr] Andriessen, J.H.M , Brok, S.W., Dekker L., Ruighaver A.B., Sips H.J.,"The Delft parallel Processor, a distributed MIMD–processror", Proc. SCS–summer Computer Conf., 1984.

[Brok] Brok, S.W, Dekker L., Kerckhoffs E.J.H., Ruighaver A.B., Sips H.J.,"Architecture and programming of the MIMD structured Delft Parallel Processor", Proc. 1st Eur. Simulation Congr., 1983.

[Dek1] Dekker, L., "Applicability of Hybrid Simulation", AICA–journal[1], pp.233-243, 4, 1975.

[Dek2] Dekker, L., "Delft Parallel Processor: a missing link to the future of simulation", Proc. SIMS annual meeting, 1982.

[Dek3] Dekker, L., "Parallel Processors in Engineering Sciences", Proc. "Simulation in Engineering Sciences", pp. 221-231, North Holland, 1983.

[Dek4] Dekker, L., "Expandability of a MIMD multiprocessor system to a large size", Proc. 'Simulators', SCS Eastern Simulation Conf., pp. 157–162, 1985.

[Dek5] Dekker, L., "Methodology–based Parallel Digital Processors", Proc. "System Analysis and Simulation: Part I, Berlin (GDR), pp. 38–50, Akademie Verlag, 1985.

[Dek6] Dekker, L., "Parallelization of 'Gauss–Seidel' algorithms: an example of parallel iterative equation solving", in print, 1987.

[Frie1] Frietman, E.E.E., Joas, M.M., Steenvoorden, G.K., "The detection of Eye–ball Movements with the Eye–SISTANT", 'Theoretical and Applied aspects of Eye Movement research', pp.5–12, 1984.

[Frie2] Frietman, E.E.E., Bruggeman, F., van Spronsen, C.J., "Eyeball–Position–controlled–COmmunication System EPCOS", Microprogramming and Microprogramming 14, no.3, pp.173–175, 1984.

[Frie3] Frietman, E.E.E., Faase, J.K., ten Kate, J.H., "Eye–switch", European Patent 78.01616, University of Technology, Delft,1978.

[Frie4] Frietman, E.E.E., van Nifterick, "The application of OPTRONICS in the field of High Speed Parallel Processing", Laser & Optronics, 1987.

[Sips] Sips, H.J., "Philosophy behind the design and construction of the Delft Parallel Procesor as a simulation tool", Proc. SIMS annual meeting, 1982.

[1]AICA = Association Internationale pour leCalcul Anologique; name AICA has been changed in: IMACS = Int. Association for Mathematics and Computers in Simulation

A PARALLEL MACHINE FOR MULTISET TRANSFORMATION AND ITS PROGRAMMING STYLE

J.-P. Banâtre (*)

A. Coutant (**)

D. Le Métayer (*)

(*) IRISA/INRIA campus de Beaulieu 35042 RENNES CEDEX

(**) IRISA/CNRS campus de Beaulieu 35042 RENNES CEDEX

Abstract

One of the most challenging problems in the field of computing science today concerns the development of software for more and more powerful parallel machines. In order to tackle this issue we present a new paradigm for parallel processing; this model, called Γ, is based on the chemical reaction metaphor: the only data structure is the multiset and the computation can be seen as a succession of chemical reactions consuming elements of the multiset and producing new elements according to specific rules. We detail some examples showing the relevance of this model for AI applications. Furthermore, due to its lack of imperative features, this language can be very naturally implemented in a distributed way. We describe an implementation of Γ on a vector architecture. The advantages of this architecture is that it needs very simple processors (with two ports) and all the processors perform the same job. So we advocate the separation of the design of programs for massively parallel machines into two steps which can be verified in a formal way: the construction of a program with implicit parallelism (Γ-program) and its translation into a network of processes.

1. Introduction.

Non conventional models of computing such as cellular automata [CODD 68], neuronic models and systolic models [KUNG 82] are now investigated as means of describing highly parallel computations for a wide spectrum of applications such as image processing, numerical computations, speech understanding and, in general, artificial intelligence applications.

Machines with a significant number of processing elements begin to appear [HILLIS 85] and people may think that computing is at the dawn of the era of massively parallel machines for the

masses. In fact, we are still far from this situation because we do not know how to master these new "monsters". Let us quote P.J. Denning [DENNING 86]:

"We do not know how to program these new (massively parallel) machines.... The software barrier - the set of limitations that impedes our effective use of parallel hardware technology - is not a brick wall that can be broken down with a heavy ram. It is a deep, thick jungle through which slow progress will be achieved by constant chopping and hacking."

So it is clear that changes in programming languages are needed...in particular it should be possible to dynamically create computations, to synchronize them and to allow information exchange between them. Several languages have been proposed with this aim: in particular C.A.R. Hoare proposed a model called CSP (Communicating Sequential Processes) [HOARE 78] allowing the definition, activation and synchronization of communicating processes. However, due to the lack of an appropriate programming methodology, it appears that it is far more difficult to build concurrent programs than usual sequential programs (although there are still important advances to be made in this area as well). This is mainly due to the fact that the programmer has to mentally manage several threads of control instead of one, as usual.

Our view is that this traditional approach to parallel programming is too imperative to become a general model for concurrent programming. We believe that only very high level languages which make parallelism implicit will allow the description of massively parallel applications.The model which is proposed in this paper is based on the chemical reaction metaphor: the computation is a succession of applications of rules which consume elements of a multiset while producing new ones and inserting them in the initial multiset. The computation terminates when no rule can be applied. The application of rules is made in a non-deterministic way and a parallel interpretation of the model is straightforward. So this model (called the Γ-model) acts as a multiset transformer. The relevance of this model to program construction is shown in [BANATRE 86] and [COUTANT 86] which present a systematic method for Γ-program derivation. This point is not detailed here as we are mainly interested in machine architecture. We describe a straightforward implementation of Γ on a vector architecture which takes advantage of its high potential for parallelism. The advantages of this implementation is that processors need only two communication ports and they all perform the same job. Furthermore the correctness of this machine can be proven rather easily. So we split the construction of programs for parallel machines into two steps: the design of a program with implicit parallelism (Γ-program) and its translation into a program with explicit parallelism (with processes). The important point is that these two steps can be verified in a formal way.

The Γ-model is quickly presented in section 2. In order to show its relevance for AI applications we describe in section 3 solutions for the path-finding problem and the Minimax algorithm. Section 4 describes the structure of the Γ-machine and gives some experimental results. The conclusion discusses related works and suggests further research.

2 The Γ-model.

The Γ-model can be described as a multiset transformer: the computation is a succession of applications of rules which consume elements of the multiset while producing new elements. The computation ends when no rule can be applied. The application of rules is performed in a non-deterministic way and a parallel interpretation of this computational model is straightforward.

Data structures.
The basic information structuring facility is the multiset which is the same as a set except that it may contain multiple occurrences of the same element. Atomic components of multisets may be of type real, character, integer, tuple of type ..., multiset of type ...
Γ function.
The main feature of the model is the Γ-function which can be defined in the following way:

$\Gamma(R,A)\ (M) =$
if $\exists\ x_1, ..., x_n \in M$ **such that** $R\ (x_1, ..., x_n)$ **then**
$\Gamma(R,A)\ (\ (M - \{x_1, ..., x_n\}) \cup A\ (x_1, ..., x_n)\ )$
else M.

Function R is called the "reaction condition"; it is a boolean function indicating in which case some elements of the multiset can react. The A function ("action") describes the result of this reaction. Let us point out that if the reaction condition holds for several subsets at the same time, the choice which is made among them is not deterministic; if these subsets are disjoint the reactions can even take place at the same time. However appropriate restrictions on the definition of condition R and action A may ensure determinacy; this point is not developed here.
This is not the most general definition of Γ; actually the Γ operator can take any number of couples (Reaction condition, Action), each reaction condition indicating in which case the associated action can be applied; however most common programs can be expressed with only one couple (Reaction condition, Action), so we will restrict our discussion to this particular case.

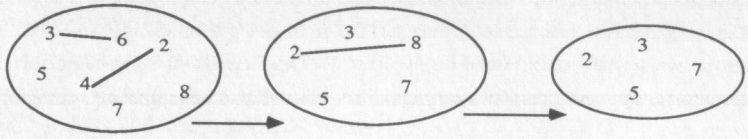
Examples of Γ-programs.
Let us now take two examples to illustrate the programming style entailed by the Γ model.

Example1.
The sieve of Eratosthenes can be written in a concise and elegant way using Γ:

sieve (n) $= \Gamma(R,A)\ (\{2, ..., n\})$ where
R (X1, X2) = multiple (X1, X2)
A (X1, X2) = {X2}

where multiple (X1, X2) is true if and only if X1 is a multiple of X2.

The following figure describes the computation of sieve (8). Of course this is one among the possible paths leading to the stable state.



Example 2

Let us consider a sorting program. The data structure is a multiset M of couples (index, value) where the index X.i of an element X gives the position of the value X.v in a virtual sequence of values to be sorted. All indexes are different and in a range [1, ..., n] where n = Card(M). The Γ-program is the following :

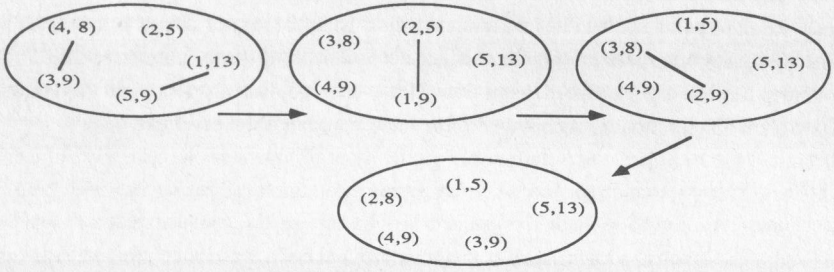
sort (M) = Γ (R,A) (M) where

R (X, Y) = (X.i > Y.i and X.v < Y.v)

A (X, Y) = { (Y.i, X.v) , (X.i, Y.v) }

The following figure describes the computation of:

sort ({ (4, 8) , (2, 5) , (3, 9) , (1, 13) , (5, 9) }) :



The stable state is reached as no couple of elements verifies the reaction condition. Of course this is only one of the possible sequences of configurations leading to the result.

This program can be seen as a generalized form of the exchange sort algorithm as any couple of ill-ordered elements can exchange their positions at any time.

3 Programming two AI problems.

In order to show the relevance of the Γ-formalism to AI applications, we consider two applications : an example of path-finding and the well-known Minimax algorithm.

3.1 Path-finding example.

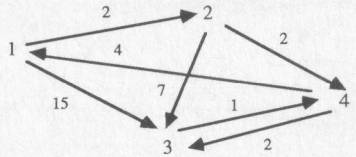
Let us consider a directed graph in which a nonnegative cost is associated with each edge. The purpose of the algorithm we describe here is to find the shortest path from a particular vertex r of the graph (called the root) to each vertex v. The shortest path is defined to be the path with lowest cost, and the cost of a path is the sum of the costs of its edges.

The graph may be represented by a multiset of tuples (i, j, c, l) where

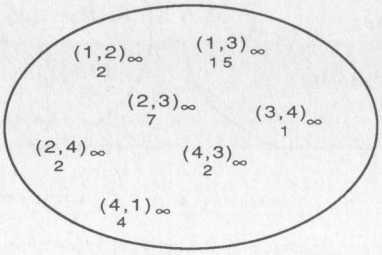
- i and j are vertex identifiers such that (i, j) is an edge of the graph,
- c is the cost of the edge (i, j),
- l is the length of the shortest known path from the root r to vertex j via vertex i.

In the following diagrams, we represent these tuples as $(i, j)_{\substack{l \\ c}}$

Considering the directed graph :



the initial multiset M_0 representing this graph is :



The following program computes the shortest path rαij, where α denotes a sequence of vertices, for each edge (i,j) of the graph :

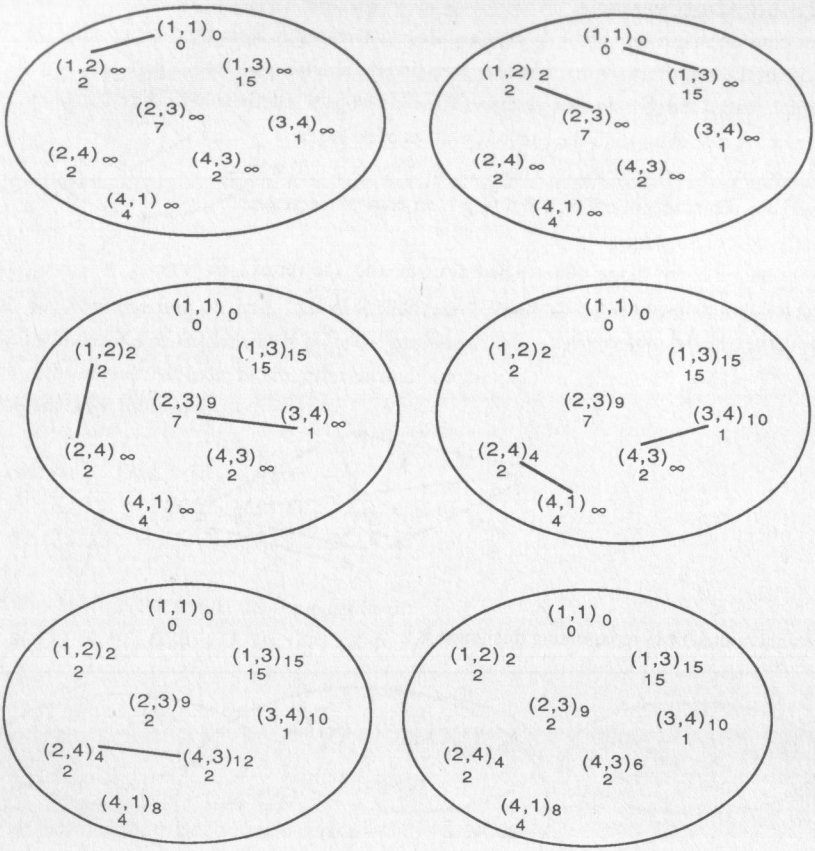
Γ(R1,A1) (M0 ∪ {(r,r,0,0)}) where

R1(X,Y) = (X.j = Y.i) and (X.l + Y.c < Y.l)

A1(X,Y) = { X, (Y.i, Y.j, Y.c, X.l + Y.c) }

A tuple $(r,r)_0$ is added to M_0 to represent the root and to ensure the start of computation.

Two tuples $(a,b)_{l1}$ and $(b,c)_{l2}$ may interact if the path rαabc of length l1 + c2 is shorter than the path rβbc of length l2. The action updates the length l2 of the shortest known path from the root to vertex c. Let us describe a possible evolution of the multiset $M = M_0 \cup \{(1,1,0,0)\}$ during the evaluation of Γ(R1,A1) (M).



The final multiset M1 may contain tuples $(i,a)_{li}$, $(j,a)_{lj}$ where $i \neq j$. In order to define the shortest path,it is necessary to keep only the tuple $(k,a)_{lk}$ with lk = Min lj s. t. $(j,a)_{lj} \in$ M1.

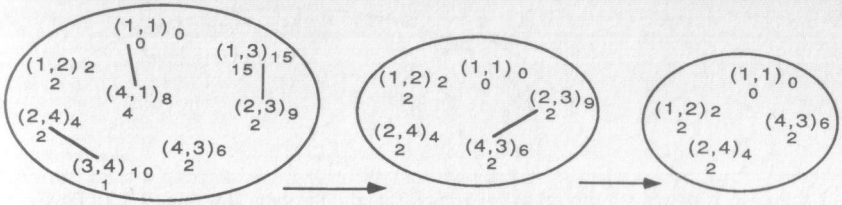
The following Γ-program performs the appropriate computation :

$\Gamma(R_2,A_2)$ (M) where

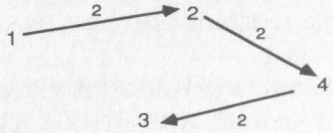
R_2 (X,Y) = (X.j = Y.j) and (X.l ≥ Y.l)

A_2 (X,Y) = {Y}

The figure below gives a possible evolution of M1 during the evaluation of $\Gamma(R_2,A_2)$(M1).



The final state of M represents the tree of the shortest paths from vertex 1 to vertices 2,3 and 4.



We have defined a program which computes the tree of the shortest paths (if any) from a given root r to the vertices in a graph. This program is a composition of two Γ-programs :

shortest_paths $(r, M_0) = \Gamma(R_2,A_2)\,(\,\Gamma(R_1,A_1)\;(\,M_0 \cup \{\,(r,r,0,0)\}\,)\,)$

$R_1(X,Y) =$ (X.j = Y.i) and (X.l + Y.c < Y.l)

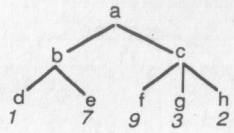
$A_1(X,Y) =$ { X, (Y.i, Y.j, Y.c, X.l + Y.c) }

$R_2(X,Y) =$ (X.j = Y.j) and (X.l ≥ Y.l)

$A_2(X,Y) =$ {Y}

3.2 Minimax example.

Minimax is a technique for searching game-trees, in order to determine the best move in a given position of a game with two adversaries [WINSTON 84]. The nodes in a game-tree represent board configurations, and they are linked by branches representing the possible moves between them. The algorithm starts with a partially developed lookahead game-tree in which the leaves are associated with advantage-specifying values. Suppose that positive numbers indicate favour to one player, called the maximizing player (the player wanting to maximize the score), and negative numbers indicate favour to the other, the minimizing player. The maximizing player is looking for a path leading to the largest positive score assuming that his opponent will always make the best choice (thus minimizing the score). So the value of a node is evaluated by taking the minimum or the maximum of the values of its sons, according to the level of node. Working up from the leaves, the values of the possible moves in the initial state are computed, and the best move is finally determined. Let us take an example of a game-tree :



Maximizing level
Minimizing level

In this case, b should get the value 1 = min (1,7) and c the value 2 = min (9,3,2). Finally a gets the value 2 and c represents the best move for the player.

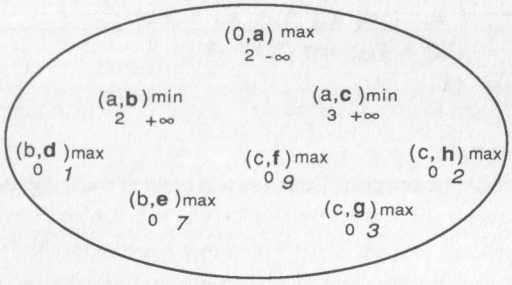
Let us now express the MINIMAX algorithm in the Γ-formalism. The data structure must represent the nodes of the tree, the relationships between them, their level and their value. These requirements lead us to define a multiset of tuples (ident, father, level, nsons, cval) where

- ident is the identifier of the node,
- father is the identifier of the father of the node; so (father, ident) is a branch of the tree,
- level is the function max or min corresponding to the level of the node,
- nsons is the number of sons of the node,
- cval is the current value of the node.

We represent such a tuple in a graphic way by : (father, **ident**) level

nsons *cval*

Let M be the multiset corresponding to the previous game-tree, its representation is the following:



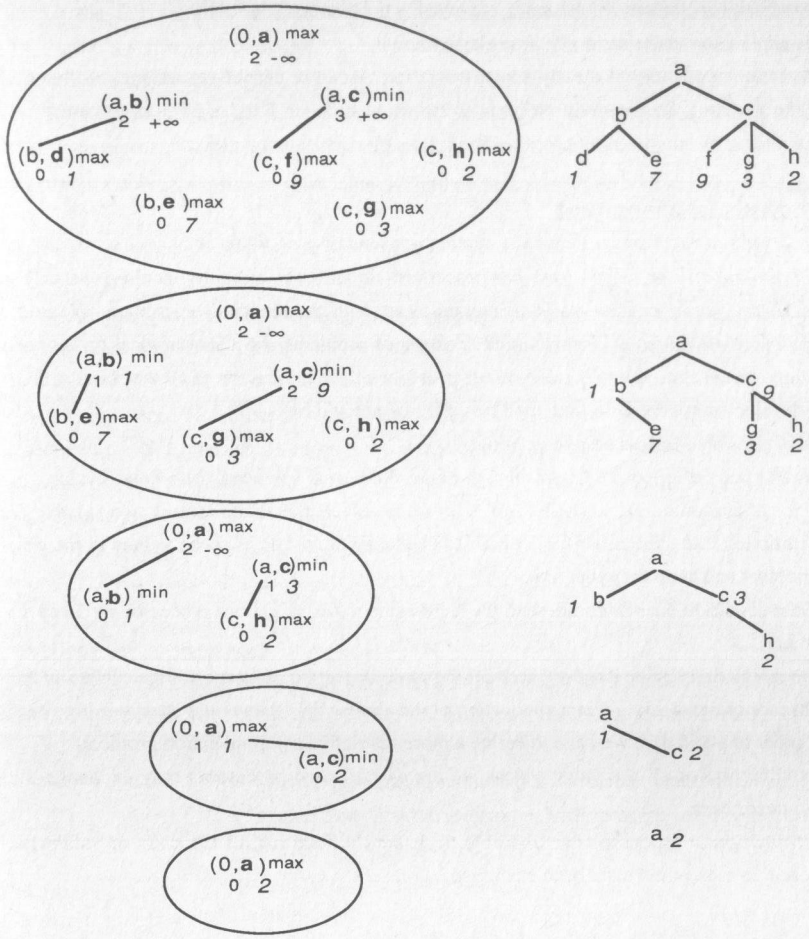
MINIMAX can be defined in Γ by :

Minimax (M) = Γ (R,A) (M) , where

R(X,Y) = (X.ident = Y.father) and (Y.nsons = 0)

A(X,Y) = {(X.ident,X.father,X.level,X.nsons-1,X.level(X.cval,Y.cval))}

Two tuples X and Y may interact if the node represented by X is the father of the node represented by Y, and the node represented by Y is a leaf of the current tree. The action removes the tuple Y and updates the tuple X : X has one child less as Y is removed from the tree, and the value of X is replaced by the maximum or the minimum of the values of the nodes. The following figure describes one possible evolution of the multiset during the evaluation of Minimax (M).



So the value we are looking for, in this case the value 2, is the cval field of the unique element of the result.

4. The Γ machine

It has been claimed in the first section that the Γ-formalism is well suited to a parallel evaluation scheme. We are now going to justify this statement by describing a parallel machine for the computation of Γ-programs. For the sake of briefness, we only consider the simplest form of programs with one binary condition and one binary action, and suggest a straightforward generalization of the method.

The evaluation of a Γ-program involves two different kinds of tasks:

-the search for elements of the multiset verifying the reaction condition,
-the application of the action to these elements.
The action application is clearly a local operation : it can be carried out independently of the rest of the multiset. So the main problem to tackle in order to build a parallel architecture is the distribution of the matching process (finding couples which may interact).

4.1 Sketch of the method

We consider in a first step that the system consists of a network of N processors and always contains exactly N values (one per processor). In order to enhance locality, we enforce the restriction that a processor may only communicate with its immediate neighbours. We exclude the duplication of values as it entails tricky coherence problems and a loss of locality. So, the values have to move through the network in order to allow the examination of all the pairs. A first sketch of the algorithm iteratively executed by a processor could be :
-receive a value from a neighbour processor,
-test the reaction condition on the couple made of the local value and the received value,
-if the condition holds, apply the action on these values; the action returns two values (since, by assumption, the system always contains N values); send one of these values to the neighbour processor and keep the other one,
-if the condition does not hold, send the local value to the neighbour processor and keep the other value.
This rough description does not account for some important properties of the system such as :
-absence of deadlock: a direct application of the above algorithm would clearly lead to deadlocks. In order to avoid this, we have to define a more restrictive communication protocol,
-examination of all the pairs: unless we are careful, two processors may exchange the same values for ever,
-termination: a processor must be able to detect the fact that all the pairs of values currently present in the system have been examined.

4.2 Topology of the network.

We describe now how these properties can be ensured very naturally using a vector topology. In order to ensure that all the pairs of values are examined, we attach a direction to each value and impose that a value moves according to its direction. When a value reaches one end of the network, its direction is reversed. This is the only case where its direction may be changed. It is easy to see that this restriction entails that all couples will be examined after a finite number of exchanges (actually N(N-1)/2). The direction can be used to avoid deadlocks as well : a processor is allowed to communicate only with the neighbour indicated by the direction of its value; moreover the communication protocol between P_i and P_{i+1} is the following : P_i sends its value to P_{i+1}; P_{i+1} computes the condition and, if necessary, the action; then P_{i+1} sends a value back to P_i. Of course P_i and P_{i+1} may communicate only if the direction of the value of P_i is True (which means "upwards"), and the direction of the value of P_{i+1} is False ("backwards"). The termination property is a bit more complicated; this problem is usually solved in distributed

systems by the introduction of a specific termination token. When a processor detects termination, the token is sent throughout the network to stop all the processors. This technique presents two main drawbacks :

-it is a special treatment which must be taken into account in the correctness proof of the algorithm. In particular, the proof of absence of deadlock is more complicated,

-it introduces a delay between the time when termination is detected and the time when it is effective.

In contrast, we present a method where each processor is responsible for its own termination : according to its local values, a processor can detect that it is no longer useful and stop; let us notice that other activities may still be going on in other parts of the network : the only property to ensure is that there will be no attempt to communicate with the stopped processor. Intuitively a processor may stop when its value has been exchanged with N-1 values, each of which had already been exchanged N-1 times at least. This means that each value in the system has been exchanged with the N-1 other values; so the reaction condition has been tested on all couples of values and no more reactions can occur. So, we associate each value V with two indicators N_1 and N_2 defined as :

-N_1 = number of moves (or exchanges) of the value.

-N_2 = number of exchanges of the value with values V' such that $N_1' \geq N-1$.

4.3 Behaviour of a processor

Let us now describe in a more formal way the behaviour of a processor i. To this aim we use the language CSP ([HOARE 78]). We just recall the most important constructs of the language :

P ? V	where P is a process name and V a variable name, is the input of a value from P. After execution, V denotes the imported value.
P ! E	where P is a process name and E an expression, is the output of the value of E to the process P.
$[GC_1 \,[]\, ... \,[]\, GC_n]$	where GC_1, ..., GC_n are Guarded Commands, is an alternative command : it executes one arbitrarily selected executable guarded command.
$*[GC_1 \,[]\, ... \,[]\, GC_n]$	where GC_1, ..., GC_n are Guarded Commands, is a repetitive command : it executes one executable guarded command (arbitrarily selected) until all guards fail.

We assume that we have N processes P(1), ..., P(N) (one per processor); the algorithm executed by an intermediate process is the following :

local variables :

int V : value.

bool D : direction of the value.

int N_1, N_2 : indicators.

int N : total number of processors.

int V', N_1', N_2' : working variables.

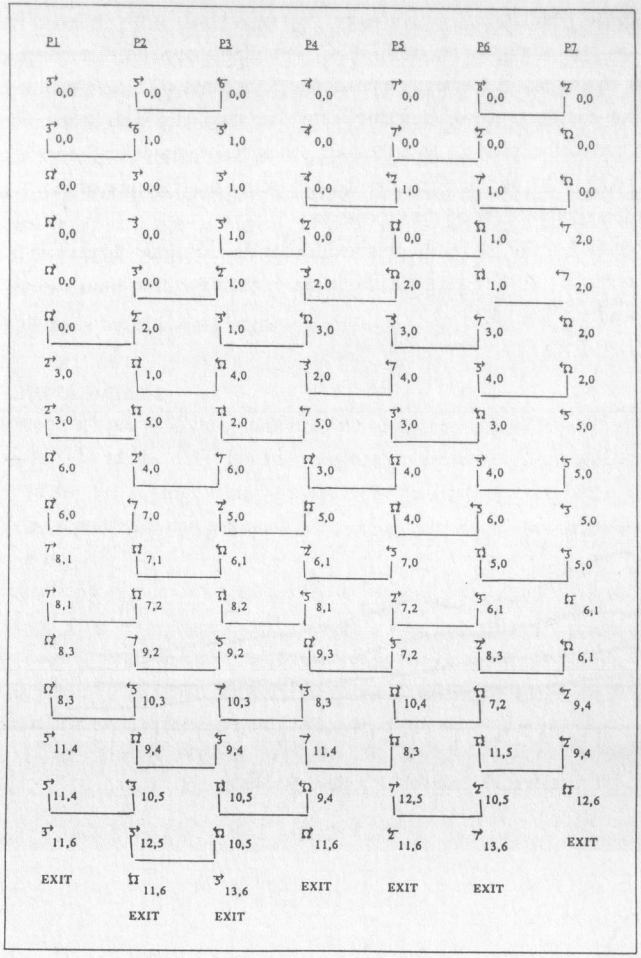
program:

```
*  [  N2 = N-1                  →  exit
   ||  N2 ≠ N-1 and D           →  P(i+1) ! (V, N1, N2) ;
                                   P(i+1) ? (V, N1, N2) ;
                                   D := F
   ||  N2 ≠ N-1 and ⌉D          →  P(i-1) ? (V', N1', N2') ;
                                   [ R(V, V')  →  {V, V'} := A(V, V') ;
                                                  (N1, N2, N1', N2') := (0,0,0,0)
                                   ||
                                     R(V',V)   →  {V, V'} := A(V',V) ;
                                                  (N1, N2, N1', N2') := (0,0,0,0)
                                   ||
                                   ⌉R(V, V') ∧ ⌉R(V',V)  →
                                                (N1, N1') := (N1 + 1, N1' + 1);
                                                [ N1 ≥ N - 1 ∧ N1' ≥ N - 1  →
                                                      (N2, N2') := (N2+1, N2'+1)
                                                || ⌉(N1 ≥ N - 1 ∧ N1' ≥ N-1) →
                                                      (N2, N2') := (0,0)
                                                ]
                                   ]
                                   P(i -1) ! (V, N1, N2) ;
                                   (V, N1, N2) := (V', N1', N2');
                                   D := T
   ]
```

The algorithm executed by processors P_1 and P_N are basically the same except that D is always True for P_1 and always False for P_N. Indicators N_1 and N_2 are initially equal to zero and the directions can be arbitrarily chosen.

4.4 Example

In order to illustrate this algorithm, we describe one possible execution of the sieve example described in section 2. Directions are figured by arrows over the values and indicators N_1 and N_2 are represented in index position. Communicating processors are underlined. The symbol Ω denotes the dummy value (or absence of value): the only particularity of this value is that it cannot react with any other value. In this example a dummy value is generated when a reaction takes place since a reaction produces only one element.



This figure represents snapshots of one among many possible computations of this system. We can however notice a few interesting points about this example :

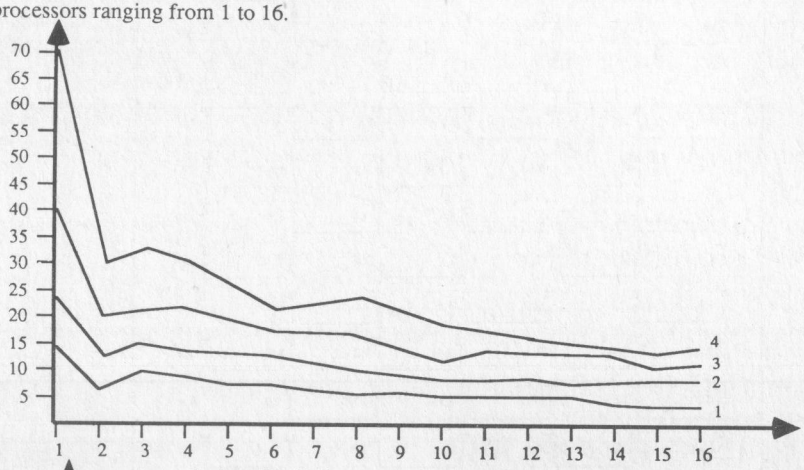
- the system has a self-balancing property : although we have chosen arbitrary initial directions, the system organized itself so that, after some steps, all inner processors are always involved in a communiaction. This property is true whenever the tasks of each processor (execution of R and A) take the same time.
- there is no deadlock in the system : this property can be formally proved.
- the termination test works correctly : when a processor stops, no neighbour will attempt to communicate with it.

The correctness of the system has been proved; we do not give details of this proof here for the sake of briefness.

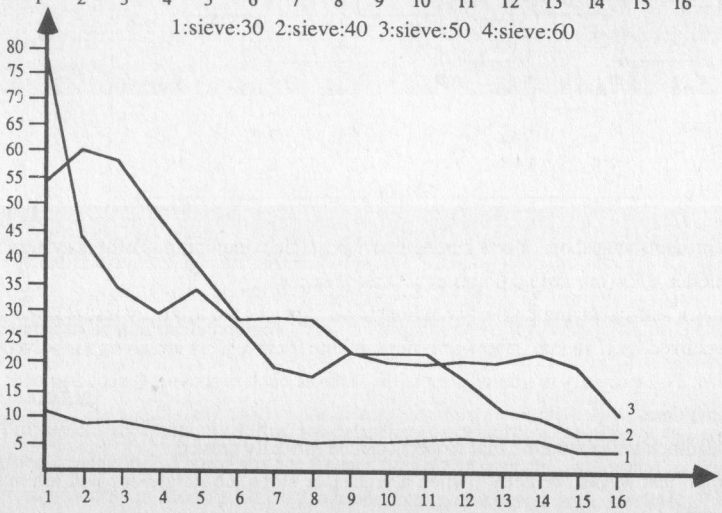
4.5. Experimental results.

This algorithm has been implemented on an IPSC (Intel Personal Super-Computer) hypercube architecture with 16 processors (currently extended to 32 processors). The machine is used in a chain configuration and the processes described above are associated with node processors. The cube manager achieves the interface with the user, loads the programs on the node processors, and active them. A graphic interface provides the user with a general view of the system (active processors, position of the values on the processors ...).

The following figures describe the results obtained for the sieve of Eratosthenes and the exchange sort described in section 2: they give the evolution of the execution time with a number of processors ranging from 1 to 16.



1:sieve:30 2:sieve:40 3:sieve:50 4:sieve:60



1: sort:<1,2,3,4,5,6,7,8,9,10,11,12,13,14,15>

2: sort:<15,14,13,12,11,10,9,8,7,6,5,4,3,2,1>

3: sort:<7,3,1,2,15,10,18,5,11,13,4,8,6,12,17,14,9,6>

These results show that Γ is well suited to a parallel implementation: with 16 processors the execution time can be divided by a factor ranging from 5 to 12 for expensive computations. Of course the performance improvement is lesser for smaller examples. We can notice some irregularities in the figures: the execution time may slightly increase when one processor is added and it is divided by more than 2 when the number of processors is doubled in the second sort experiment. This is due to the fact that the couples of values are not examined in the same order when the number of processors changes. So, by chance, one processor more can dramatically decrease or slightly increase the execution time. The results presented here correspond to the first naïve implementation of Γ. We describe in the next section some simple modifications which can improve significantly the efficiency of the system.

4.6 Some improvements

The correctness proof of the system involves a property which states that two values with indicators $N_1 \geq N\text{-}1$ and $N'_1 \geq N\text{-}1$ have already been exchanged; this means that R has already been evaluated for these values and turned out to be false as these values have not been transformed. So a processor can detect such a case (as it possesses the value of N) and avoid a reevaluation of R.

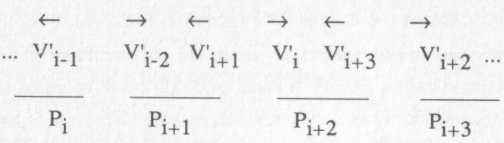
A second improvement is suggested by the example described in section 3.4. We have claimed that the system organizes itself in such a way that all inner processors are always involved in a communication. This means that one processor out of two is computing a reaction condition and, possibly, an action while the other one is waiting for the result. We can clearly do better; the situation can be described by the following figure :

$$\begin{array}{cccccccccccc} & & \rightarrow & & \leftarrow & & \rightarrow & & \leftarrow & & & \\ \cdots & V_{i-2} & & V_{i-1} & & V_i & & V_{i+1} & & V_{i+2} & V_{i+3} & \cdots \\ & \overline{\quad} & & \overline{\quad} & & \overline{\quad} & & \overline{\quad} & & \overline{\quad} & \overline{\quad} & \\ & P_{i-2} & & P_{i-1} & & P_i & & P_{i+1} & & P_{i+2} & P_{i+3} & \end{array}$$

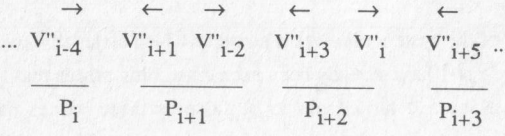
Processors P_{i-1}, P_{i+1}, and P_{i+3} are computing, and P_{i-2} , P_i and P_{i+2} are waiting. An obvious improvement consists in associating two values V_i V_{i+1} with each processor. The basic cycle of a processor would be to send V_{i+1} to the right neighbour, to wait for a value from the left neighbour and to apply R (and possibly A) to this value and the local value V_i. Then a processor has to send its result to the left neighbour, wait for the result of the right neighbour and apply R (and possibly A) on its two local values. In other terms, a processor would behave exactly as a couple of neighbour processors in the previous system. This is illustrated by the following figure:

$$\begin{array}{cccccccccccc} & & \rightarrow & & \leftarrow & & \rightarrow & & \leftarrow & & & \\ \cdots & V_{i-2} & & V_{i-1} & & V_i & & V_{i+1} & & V_{i+2} & V_{i+3} & \cdots \\ & \overline{\quad} & & \overline{\quad\quad} & & & & \overline{\quad\quad} & & & \overline{\quad} & \\ & P_i & & P_{i+1} & & & & P_{i+2} & & & P_{i+3} & \end{array}$$

P_{i+1} sends V_i to P_{i+2} and receives V_{i-2} from P_i. It computes $R(V_{i-2}, V_{i-1})$ (and possibly $A(V_{i-2}, V_{i-1})$), returns a result V'_{i-1} to Pi ($V'_{i-1} = V_{i-1}$ if no reaction occurs), and receives V'_{i+1} from P_{i+2}.



Each processor computes its local reaction :



This modification has a drastic effect on the efficiency as, in the best case, it allows us to halve execution time, or to achieve the same performance with half the number of processors.
We have just described here the simplest case where the number of values is equal to the number of processors (or the double of it with the latter improvement). The general case can be implemented in the very same way by considering sets of values instead of values. A more difficult extension concerns N-ary conditions and actions, because we have to examine all subsets of N elements. The text of the processors in these cases is a bit more intricate but the basic idea is quite simple : in order to examine all the subsets of size N, we have to examine all the pairs of one element and one subset of size N-1. So we can reduce the general case to the binary case and use the same technique as before.

5.Conclusion.

We have intoduced in this paper a parallel machine for the implementation of a high-level formalism called Γ. It is interesting to compare our work with the Dijkstra-Gries approach to iterative program construction. Let us first translate the simplest form of loop into a Γ-program. In the Dijkstra-Gries formalism ([DIJKSTRA 76], [GRIES 81]), a simple loop looks as follows:

*[C(x) ----> Ψ(x)]

It expresses that the loop will go on applying Ψ to the variable x while C(x) remains **true**. It can be translated into the following Γ-program:

$\Gamma(C', \Psi')$ where
$C'(y)=C(y)$
$\Psi'(y)=\{\Psi\ (y)\}$

The reaction condition expresses that if the unique element y of the argument multiset is such that C(y) then it is replaced by $\Psi(y)$.
More generally, a loop composed of n guarded commands and operating on k variables denoted by the vector $X=(x_1, ..., x_k)$ may be written as:

$$
\begin{aligned}
&*[\ C_1(X) \text{ ---> } \Psi_1(X)\\
&\quad ...\\
&\quad C_n(X) \text{ ---> } \Psi_n(X)\\
&]
\end{aligned}
$$

and translated into:

$\Gamma((C'_1, \Psi'_1),\ldots ,(C'_n,\Psi'_n))$ where
$C'i(y)=Ci(y)$
$\Psi'i(y)=\{\Psi i\ (y)\}$

In these examples, the state of computation is represented by the state of the variables. The net effect of executing a statement Ψ_i is to change the values of the variables. Of course, no parallelism is possible as in both cases the argument multiset possesses only one element. There is no simple way of allowing several actions to operate simultaneously on disjoint subsets of the global state. This could only be achieved by **explicitly** programming the constitution of multisets and their modification. Even this solution would imply very important changes in the semantics of the usual guarded commands. Furthermore, the data structuring facilities (arrays) used in conjunction with guarded commands entail a sequential style of programming. It is clear that more descriptive and more general data structures are necessary in order to construct programs with a high potential for parallelism. The interested reader may refer to ([BANATRE 87], [COUTANT 86]) for the description of a systematic method for the derivation of Γ-programs.
The non-imperative features of the language can be taken into profit in a distributed implementation. The parallel architecture defined to execute this language is a vector of identical processors. The main advantage of this network is that it allows the use of very cheap processors (only two ports). Furthermore the correctness proof of this implementation can be established in a rather simple way. So we decompose the construction of programs for massively parallel machines in two steps:
- the construction of a program with implicit parallelism.
- the derivation of a program with explicit parallelism.

The important point here is that these two steps can be formally proven. Of course work has still to be done in these two areas, mainly as far as efficiency is concerned; we have to find

optimization rules allowing the system to take advantage of the logical properties of programs to improve their implementation. Several promising projects concerning the Γ-model have been undertaken, let us mention: extension of the formalism to provide more powerful data-structuring facilities, systematic program derivation and mechanization of proofs of properties of Γ-programs.

REFERENCES

[BANATRE 86] Banâtre J.-P. and Le Métayer D., A new computational model and its discipline of programming, INRIA research report N° 566, Sept. 86.

[BANATRE 87] Banâtre J.-P. and Le Métayer D., A new approach to systematic program derivation, in Proceedings of the 1987 Workshop on Software Specification and Design (Monterey April 3-4), IEEE pp. 208-215.

[CODD 68] Codd E. F., Cellular Automata, Academic Press, 1968.

[COUTANT 86] Synthèse de programmes dans le formalisme Γ, M.S.C. Thesis, June 1986.

[DENNING 86] Denning P. J., Parallel computing and its evolution, Comm. ACM 29,12 (December 86), pp. 1163-1167.

[DIJKSTRA 76] Dijkstra E. W., A discipline of programming, Prentice-Hall, Englewood Cliffs, N.J., 1976.

[GRIES 81] Gries D., The science of programming, Springer-Verlag, 1981.

[HILLIS 85] Hillis W. D., The connection machine, MIT press, Cambridge, Mass., 1985.

[HOARE 78] Hoare C. A. R., Communicating Sequential Processes, Comm. ACM 21, 8 (August 1978), pp. 666-677.

[KUNG 82] Kung H. T., Why Systolic Architectures ?, Computer Magazine 15,1 (January 1982), pp. 37-46.

[WINSTON 84] Winston P. H., Artificial Intelligence, Addison-Wesley, 1984.

CELLULAR ARRAY ARCHITECTURE FOR RELATIONAL DATABASE IMPLEMENTATION

Mieczyslaw R. Muraszkiewicz

Institute for Scientific, Technical
and Economic Information
00-041 Warsaw, ul. Jasna 14/16, Poland

The design of high performance database machine architectures is a process wherein the mutual interaction of algorithms, data structures, and technology determines the final result. In the paper fully parallel algorithms for implementing the relational algebra operators are presented. Special emphasis is put on SELECT, PROJECT and JOIN. A relational database machine architecture based upon cellular arrays is proposed. A straight mapping between the relation structure and cellular array geometry, and the fact that cellular arrays can be easily implemented on a commercial scale as VLSI chips are the rationales of this proposal.

1. INTRODUCTION

Undoubtedly, the von Neumann paradigm for serial data processing has already lost its inspiring drive in the community of computer architecture designers. The new technology makes it feasible to build low-cost special purpose devices in which thousands or even tens of thousands processors cooperate to solve sophisticated problems. However, the question arises: now that we can put essentially anything on a chip, what ideas are worth putting on a chip? We believe that parallel nonnumeric data processing, in particular relational databases and knowledge information systems equipped with inference engines are relevant candidates to be put on chips.

In this paper we are interested in parallel associative nonnumeric data processing; more precisely in relational database computers. Let us formulate some postulates for one who wishes to design the architecture of a relational database computer:

- highly parallel structures composed of a large number of possibly homogeneous computing elements organized in a regular pattern should be employed. In particular, SIMD and MIMD architectures (cf. Flynn [6]) seem to offer the desired features;
- associative memories (processors) rather than conventional ones have to be taken into consideration;
- software should be substituted by hardware;
- data storing and data processing should be performed at the same place in order to minimize data transmissions.

We claim that cellular arrays constitute a good basis to determine the approach satisfying the above-mentioned postulates [8,13,14]. This is because there exists a straight mapping between the

relational model of data and logical structure of cellular arrays which are rectangular patterns composed of primitive processors. Moreover, cellular arrays offer a number of advantages: logical flexibility, ease of logical and circuit design, ease of testing and diagnosis, fault avoidance, high package denesity, low power level and high speed, ease of functional decomposition of a system into chip-size "modules", increased parallelism at all levels and facilitated manufacturing as VLSI. In general, the cellular arrays appear to offer a means whereby the tremendous improvements in overall performance promised by VLSI may actually be achieved. To be quite honest about the issue we have to say that a major implementation constraint of large networks in the VLSI environment is the number of pins available on a chip. Some suggestions how to overcome the pin limitation constraint can be found in the paper by Franklin [7].

Several papers have been written about cellular arrays for nonnumeric data processing. The reader interested in this subject can refer to Berra [1], Deverell [4], Fialkowski [5], Kautz [8,9], Muraszkiewicz [13,14,15], Ramamoorthy [16].

An outline of this paper is as follows. In the next Section we introduce the notations concerning relations and cellular arrays. In Section 3 we shall show how SELECT, SORT, PROJECT, CARTESIAN and JOIN can be implemented in cellular arrays. At the end of the paper we shall make some remarks on data integrity checking, expert systems and physical realization of cellular arrays.

2. PRELIMINARIES. CELLULAR ARRAY

In the relational model, Codd [2,3], the data are viewed as a collection of non-hierarchical time-varying relations. Relations may be thought of as being stored in tables such that each row in the table has the same format. The columns of the table contain the values of the attributes and the rows store the tuples which describe some objects or relationships in the real world. All the rows must be distinct. A relation is invariant under permutation of rows; however in the next sections we shall very often prefer to operate on the ordered relations (tables) in which the tuples are stored. Usually, it is assumed that the ordering of the columns within a table is immaterial. Later on, this will not be the case: we want a sequence of columns to be fixed during processing.

Let $R(A_1,\ldots,A_k)$ be a relation whose name is R, with the attributes $A_1,\ldots,A_k$. We assume that $r_i=(a_{i1},\ldots,a_{ik})$ is a tuple from the i th row of R, where a_{ij} belongs to the domain of A_j, $j=1,\ldots,k$. In this paper we restrict our discussion to the simple relations only where all the a_{ij}'s are single elements, i.e. relations which are normalized (1NF,2NF,3NF or 4NF). Physically, a tuple $r=(a_1,\ldots,a_k)$ can be represented as a word r which is a concatenation of subwords $a_1,\ldots,a_k$. The total length of r equals the sum of a_i's lengths, $i=1,\ldots,k$. Let this sum be n. We assume that all the tuples in R are of the same length n. Thus, the relation R composed of m tuples can be regarded as a table of characters, denoted by R[m,n], made up of m rows and n columns. An

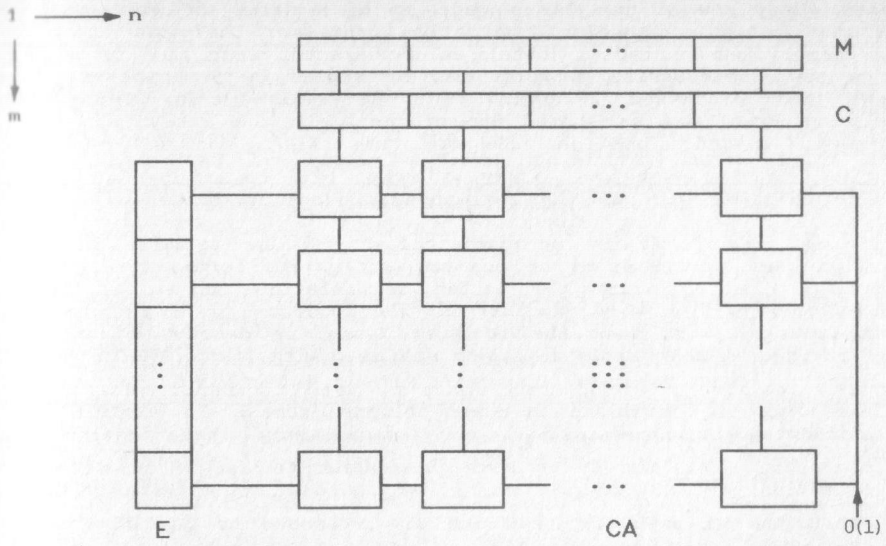


Fig.1. Cellular Array structure

ij element of R[m,n] is referred to as r[i,j]. For convenience, despite the fact that the relation R may be changed as a result of some manipulations, we shall preserve its original name during and after processing.

Let us briefly present a cellular array which is proposed to handle relations. A network of identical elements called cells which are organized so that they form a rectangular homogeneous and regular array is reffered to as CA (Cellular Array). Fig.1 displays the CA structure wherein one can distinguish m*n cells arranged in m rows and n columns. The figure includes some registers that will be presented below. The array rows are numbered downwards from 1 to m whereas the columns are numbered from 1 to n, from left to right. Boxes represent cells, lines denote cells' interconnections. Each cell can be connected to its east, south, west and north neighbours, and the cell around the edges of the array are connected to either auxiliary registers which lie outside the array proper or to fixed external signals.

A cell is assumed to be able to perform the following operations: (1) store one character; (2) set up its status - either transparent or opaque; (3) compare two characters, matching included; (4) move a character to its south neighbour; (5) interchange two characters from the adjacent cells in a column; (6) read-out a character from its memory; (7) write a character into its memory; (8) transfer a bit from its east input to its west output, if a transparent status is set up. A particular operation to be performed is determined by a control signal delivered to the cell from an external control unit (not shown in Fig.1). All the cells are supposed to be supplied with the same control signal, thus they execute the same operation at a time. In other words, CA works as a SIMD machine.

Since every row of the CA is made up of n cells and each cell is assumed to be a primitive associative processor that can store in its memory one character (byte), an n-character word may be stored in a row. Consequently, m words may be stored in CA composed of m rows and processed according to the cells' logic. Thus, the relation R[m,n] may be stored and processed in the array. To get a straight mapping between the relation $R(A_1,...,A_k)$ and CA, we assume that consecutive array columns are collected in k the so-called zones each of which corresponds to one attribute.

Different registers can be connected to CA. A register can be regarded as a one-column or one-row array. The former is called a vertical register while the latter is referred to as horizontal one. Note, that a word (vector) is the contents of a register. In the sequel a word from the register X will be denoted by x. Now, let us present some registers which are used with CA (cf. Fig.1):

- mask register M (horizontal) which is used to inhibit the execution of operations in some columns (zones). To exclude the attributes $\alpha_1,..,\alpha_l \in \{A_1,...,A_k\}$ one can apply the $MASK(\alpha_1,...,\alpha_l)$ operation. Sometimes it is more convenient to use a complementary operation $\overline{MASK}(\beta_1,...,\beta_h)$ where $\beta_1,...,\beta_h \in \{A_1,...,A_k\}$ are the attributes of one's interest and are supposed to participate in the output relation. When MASK is used we actually deal with subtuples of R; they are denoted $\underline{r}$;
- comperand register C (horizontal) which is to be employed for the implementation of SELECT and JOIN operators. We assume that a tuple from the array can be read-out to C, and vice versa. Physically, the stage output of both C and M registers are bussed vertically along columns to all rows of the array;
- word selection register E (vertical) where the result of comparison of words is injected. All the cell output terminals on the left-hand side of CA serve as inputs to the stages of E (the inputs of the right edge of CA are connected to a constant signal fixed at 1 or 0). It may happen that the register E plays the role of the characteristic register which indicates those tuples in CA which belong to the output relation..

Now, we introduce the operation of restriction, denoted RESTRICT(E,R,b), where $b \in \{0,1\}$. This operation allows to choose the tuples of one's interest. It operates on R so that all the tuples for which e[i]=b will belong to the output relation while those tuples for which e[i]=$\overline{b}$, i=1,...,m, are considered as foreign tuples. Later on RESTRICT usually follows another very handy procedure, namely the CENTRIFUGE operation which transforms a sequence of words so that it is composed of two consecutive subsequences: first, where there are only the words for which e[i]=b second which is composed of the words for which e[i] = $\overline{b}$.

3. IMPLEMENTING THE RELATIONAL ALGEBRA

Relational Algebra has received the most attention as a candidate for hardware support, e.g. Kung [10]. This algebra is composed of a group of relational operators which can be applied to original relations to create new relations. A set of relational operators includes : traditional set theory operations like Cartesian product, union, intersection, difference and less traditional operations lik selection, projection, join and division. These

operations (even a part of them) form a basis for a user query language building to communicate with a database (cf. Maier [11], Ullman [17]).

SELECT

We shall describe SELECT for the case of greater-than selection, however, it can be easily modified to cover other types of selection ($\geq,<,\leq,=,\neq$). The most significant characters of a word are assumed to occupy the leftend positions in the array row.

Given a relation R[m,n], a vector C[n] and a mask. One wants to SELECT all the words that hold the condition $\underline{c} > \underline{r}_i$, i=1,...,m. By a couple we understand a pair (c[j],r[i,j]), i=1,...,m; j=1,...,n. SELECT runs as follows:

(a) Set up MASK. All the masked couples are considerd as transparent and labelled by "t". They are excluded from processing.

(b) Within each non-transparent couple matching involving couple's elements is performed. Matchings are done in parallel in all the couples. If c[j] = r[i,j] then a couple is considered as transparent and labelled by "t"; otherwise it is called opaque and marked with "o".

(c) Predecessors are compared with successors simultaneously in all the opaque couples only. If c[j] > r[i,j] then 1 is assigned to the couple; otherwise 0 is assigned.

(d) A word selection vector E[m] is created so that the result of comparison obtained in step (c) for the most significant opaque couple (if any) from the i-th row is treated as a value e[i], i=1,...,m. If there are no opaque couples concerning this row, e[i] = 0 is assumed (recall here a constant signal fixed at 0 at the right edge of CA, cf. Fig.1). Note that for i=1,...,m:

$$e[i] = \begin{cases} 1, & \text{if } \underline{c} > \underline{r}_i \\ 0, & \text{otherwise.} \end{cases}$$

(e) CENTRIFUGE(E,R,↓) (cf. subsection on sorting)

(f) RESTRICT(E,R,1)

Example 1. Let 1<...<9<A<...<Z. Fig. 2 displays >-selection for R[4,3] and C[3] such that r_1=TOM, r_2=TED, r_3=DIC, r_4=TIM and c=TIM; masking does not occur. A superscript "t" or "o" by r[i,j] appears for transparent or opaques couples respectively, (step b), while a subscript denotes the comparison result (for opaque couples only) obtained in step (c). The most significant nonmatching positions which contribute to the vector E are marked with an asterisk. The output relation is composed of the words r_2, r_3 ■■

It is worth mentioning that the proposed method exhibits associative features and belong to the class of fully parallel algorithms (parallel-by-character, parallel-by-word). Later on, the SELECT operation is to be referred to as SELECT(C,R,>). We assume that similar operations like SELECT(C,R,=), SELECT(C,R,<), etc. are also available, if needed. If steps from (a) to (d) are performed only, we call such procedures external comparisons and

(a) (b)

$$\begin{array}{|ccc|} T & I & M \end{array} = C[3]$$

$$E[4] = \left|\begin{array}{c} 0 \\ 1 \\ 1 \\ 0 \end{array}\right| \left|\begin{array}{ccc} T^t & *O_0^o & M^t \\ T^t & *E_1^o & D_1^o \\ *D_1^o & I^t & C_1^o \\ T^t & I^t & M^t \end{array}\right| = R[4,3] \qquad \left|\begin{array}{ccc} T & E & D \\ D & I & C \end{array}\right| = R[2,3]$$

Fig.2. Example of >- selection
(a) presentation of selection running; (b) output relation

denote them EXCOMP(E,C,R,>), EXCOMP(E,C,R,=), EXCOMP(E,C,R,<), etc., respectively. Moreover, one can note that using the above algorithm one can determine internal (adjoining) word selection operations. In this case, the vector C is not required and a couple is now defined as follows: (r[i-1,j],r[i,j]), i=2,...,m; j=1,...,m. The rest of the presented algorithm does not change. The modification means that words $\underline{r}_{i-1}, \underline{r}_i$, i=2,...,m are compared rather than $\underline{c}$, $\underline{r}_i$, i=1,...,m. Such an operation is denoted INCOMP(E,R,>). Additionally, we want the operations like INCOMP(E,R,=), INCOMP(E,R,<), etc. to be at our disposal, as well. It should be noted that INCOMPs leave a value of the e[1] bit undefined.

SORT

A sorting method is drawn from Muraszkiewicz [13]. Let us recall briefly this method. Given some mask and a relation R[m,n] where one can distinguish a sequence of words *seq* = $\underline{r}_1,\ldots,\underline{r}_m$. We say that the sequence is sorted (in ascending order) if:

$$(\forall\ i,j \in \langle 1,m \rangle)\ (i < j \Rightarrow \underline{r}_i > \underline{r}_j).$$

Let $cp_i = (\underline{r}_{i-1}, \underline{r}_i)$ be a couple. We define an exchanging function

$$exch(cp_i) = \begin{cases} (\underline{r}_i, \underline{r}_{i-1}), & \text{if } \underline{r}_{i-1} > \underline{r}_i \\ (\underline{r}_{i-1}, \underline{r}_i), & \text{otherwise.} \end{cases}$$

We say that an even phase, denoted by *EP*, is performed over the sequence *seq* when the function $exch(cp_{2i})$ is realized for i=1,..,entier(m/2); an odd phase, denoted *OP*, is executed when function $exch(cp_{2i+1})$ is realized. The realization of these functions proceeds in two steps: first, INCOMP(E,R,>) is performed; second, predecessors and successors are exchanged each other in these couples where predecessors are greater than successors. It is assumed that masking does not inhibit exchanging of subwords from the masked zones.

Example 2. Given a sequence:SAM, ED2, BOB, ED1; no mask is used. Fig.3 shows how the sequence is transformed by *EP*, *OP*, *EP*, *OP* phases. The couples are taken into rectangulars; if exchanging

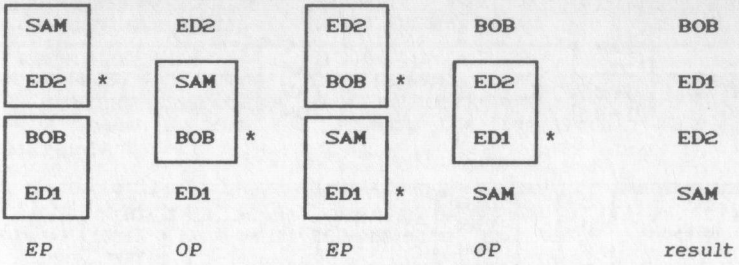


Fig.3. Example of sorting

occurs, an asterisk is added. It is apparent from the figure that the rightmost sequence is ordered ■■

Now, we may generalize the above example and claim that sorting of the sequence *seq* is to be accomplished as a result of successive executions of no more than m phases in the following way

EP, OP, EP, OP,... or *OP, EP, OP, EP,...*

where m is the number of words to be ordered. Note that sorting runs in O(m) time.

Our sorting method allows to obtain not only ascending sequences; if we redefine the *exch* function, we could obtain sorting for getting descending sequences, too. For the relation R, the former is labelled by SORT(R,$\uparrow$) whereas the latter by SORT(R,$\downarrow$).

Given a vertical binary vector E[m] and a relation R[m,n]. Let us apply SORT(E,$\uparrow$) for E and extend this operation so that every movement of the e[i] bit one row upwards or downwards is accompanied by a similar movement of the r_i word in R, i=1,...,m. We have just defined the CENTRIFUGE(E,R,$\uparrow$) operation mentioned in Section 2. In like manner one can define CENTRIFUGE(E,R,$\downarrow$).

PROJECT

The value of the PROJECTion is that relation which is obtained by selecting the indicated attributes (zones) of the original relation and removing any redundancy from the rows of the table so formed.

Given a relation R[m,n] and a set of atributes $\{\alpha_1,...,\alpha_1\}$ which are to be removed from R. PROJECT can be conducted as follows

```
MASK(α1,...,α1)
SORT(R,↑)
INCOMP(E,R,=);  Force e[1] = 0
CENTRIFUGE(E,R,↓)
RESTRICT(E,R,0)
```

SORT causes identical words (with regard for masking) to be put as neighbours; INCOMP allows to identify all duplicates and CENTRIFUGE moves all of them to the low region of the table.

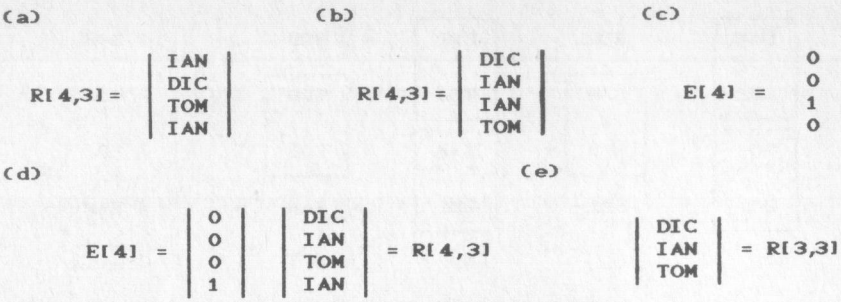


Fig.4. Example of projection
(a) input relation R, (b) relation after SORTing,
(c) vector E after INCOMP(E.R,=)ing
(d) E and R after centrifuging, (e) output relation

Example 3. Fig. 4. shows how the projection proceeds for a sample relation R[4,3] where r_1=IAN, r_2=DIC, r_3=TOM, r_4=IAN; there is no mask here. Fig. 4(a) displays the original relation; Fig. 4(b) presents the relation after SORTing; Fig. 4(c) shows the vector E[4] after INCOMPing, and Fig. 4(d) displays E and R after CENTRIFUGing. The last picture, Fig. 4(e), shows the output relation, after RESTRICTing ■■

JOIN

We shall restrict our attention to the equi-JOIN. The idea behind this operation is that we take a word from the first relation, look for identical words (with due regard for masking) in the second relation and concatenate the former with every one of these words. We repeat that for all the words belonging to the first relation.

Given the relations R[m,n] and S[k,n]. Further on, we assume that the number of rows in R and S (as well as in the arrays wherein the relations are stored) may increase in case of need during JOIN running. Before we present a method of joining in more detail, we have to introduce some auxiliary procedures, viz. COMMON, COUNT, PUSH, WCOPY, SCOPY and the CARTESIAN procedure.

COMMON(R,S) operates on the relations R, S and the registers C[n], E_R[m] and E_S[m]. Moreover, two supplementary binary registers E_R[m] and E_S[m] are assumed to be available. $E_R \vee E_R$ (or $E_S \vee E_S$) denotes logical positional sum of the registers E_R and E_R, i.e. $e_R[i] \vee e_R[i]$, i=1,...,m. COMMON, which selects all the words in R that occur in S and vice versa, can be described as follows

E_R:= E_S:= 0...0;
for i:=1 **step** 1 **until** m **do**
begin
put r_i to C;
EXCOMP(E_R,C,R,=); EXCOMP(E_S,C,S=);
E_R:= $E_R \vee E_R$; E_S:= $E_S \vee E_S$

```
    end;
CENTRIFUGE(E_R,R,↓);        CENTRIFUGE(E_S,S,↓);
RESTRICT(E_R,R,1);          RESTRICT(E_S,S,1).
```

Note, that EXCOMP operations may be performed for R and S in parallel. The same is true for summing the registers, CENTRIFUGing and RESTRICTing.

COUNT(R,c) is a function that returns the number of occurences of the word c in R. PUSH(R,i,k) is employed when one wants a sequence of words $r_i,...,r_m$ to shift downwards k times (rows). This procedure is performed to free an area inside R in order to get "empty room" for further processing. WCOPY(R,i,1) is a procedure which duplicates the word r_i and puts its copies into the rows i+1,...,i+1. SCOPY(R,i,1) copies the words $r_i,...,r_{i+1-1}$ into the rows i+1,...,i+21-1. It is left to the reader to elaborate methods for implementing COUNT, PUSH, WCOPY and SCOPY.

Now, we present a procedure which creates the CARTESIAN product of the relations R[m,n] and S[k,n], i.e. all possible concatenations of all the tuples from R and S are created. CARTESIAN runs as follows

(a)

```
i:=1;
for l:=1 step 1 until m do
    begin
      PUSH(R,i+1,k-1);
      WCOPY(R,i,k-1);
      i:= i+k
    end
```

(b)

```
i:=1;
for l:=1 step 1 until m-1 do
    begin
      PUSH(S,i+1,k);
      SCOPY(S,i,k);
      i:= i+k
    end
```

(c) R <=> S (to each r concatenate s).

Note, that steps (a) and (b) can be performed in parallel.

Let P[u,z] be a relation. A subrelation of P that is composed of tuples $p_j,...,p_k$, $1 \le j \le k \le u$, is to be referred to as P/j:k/.By subCARTESIAN(R/w:w'/, S/t:t'/) we understand the procedure which is built up of steps (a) and (b) of CARTESIAN and operates on subrelations of R and S (instead of i:=1 we have i:=w and i:=t, and m takes a value of w'-w+1 and k a value of t'-t+1).

Given the relations $R(A_1,...,A_k)$, $S(B_1,...,B_k)$ and $\overline{MASK}(A_j)$, $\overline{MASK}(B_j)$,$j \le k$. It is supposed that the domaines of A_j and B_j are identical. Additionally, we assume that zones connected to A_j and B_j are of the same length and starts from the CAs' columns having the same number. JOINing of R[m,n] and S[k,n] proceeds as follows

```
MASK(A_j); MASK(B_j);
COMMON(R,S);
SORT(R,↑); SORT(S,↑);
i:=1;
while tuple r_i belongs to R do
      begin
```

```
            put r_i to C;
            u:= COUNT(R/r_i); w:= COUNT(S/r_i);
            if u≠1 ∨ w≠1
               then begin
                      subCARTESIAN(R/i:i+u-1/,S/i:i+w-1/);
                      i:= i+u*w
                    end
               else i:= i+1
         end
   R <=> S.
```

If A_j and B_j are identical, one of them may be removed; this case is called a natural join.

(a)

R	S
B1	A3
A1	A4
A2	A5
C1	B2
	D2

(b)

R	S
B1	A3
A1	A4
A2	A5
	B2

(c)

R	S
A1	A3
A2	A4
B1	A5
	B2

(d)

R	S
A1	A3
A1	A4
A1	A5
A2	A3
B1	A4
	A5
	B2

(e)

R	S
A1	A3
A1	A4
A1	A5
A2	A3
A2	A4
A2	A5
B1	B2

Fig.5. Example of joining (digital position is supposed to be masked)
(a) input relations R, S, (b) R, S after COMMONing,
(b) R, S after sorting,
(c) R, S after first looping within CARTESIAN,
(e) output relations (before concatenation)

Example 4. Fig.5 shows how JOINing proceeds for two sample relations. The positions occupied by digits in tuples are supposed to be masked ■■

4. FINAL REMARKS

Fully parallel methods for implementing the most significant relational algebra operators in cellular arrays and the structure of these arrays have been proposed in the paper. Thus, a basis for designing the detailed architecture of a relational database computer has been established. Now, let us supplement our considerations with three remarks:

(1) Problems of semantic integrity were not discussed in the paper. Usually, the DBMS is responsible for checking certain kinds of consistency constraints (required properties of data). It seems that cellular arrays might be charged with checking restrictions that depend on the semantics of domains (attribute) elements and restrictions on relations that depend only on equality or inequality of values (functional dependencies).

(2) To evaluate the possibilities of the CA implementation as a VLSI chip we applied the Conway-Mead approach [12]. It turned out

that for a basic metric λ = 1μm and a chip 9*9 mm^2 area one could place on it about 10^5 cells (with wiring and additional devices like testing circuitry). The fact that a linear decrease in λ leads to approximately an λ^2 improvement in packaging denesity is a good prognostic for cellular arrays.

(3) Combination of relational databases with inference (deductive) mechanisms is a possible approach to design expert systems. The specific knowledge (facts and rules) can be stored in cellular arrays. The operators and auxiliary procedures presented in the paper may be helpful for designing inference operators.

Some problems have not been discussed in the paper. For instance, the problem that the size of a relation exceeds the size of the cellular array. Another question is whether the proposed network is the only one for implementing relational databases. How can one design a cellular array oriented language including parallelism? All these problems, and many other issues, are discussed in detail in Muraszkiewicz [14].

REFERENCES

[1] Berra P.B., Oliver E., The Role of Associative Array Processor in Data Base Machine Architecture, Computer, 3 (1979) 53-61.

[2] Codd E.F., Further Normalization of the Data Base Relational Model, in: R.Rustin, (ed.),Courant Computer Science Symposium 6 - Data Base Systems (Prentice-Hall Inc., 1972) 33-64.

[3] Codd E.F., Relational Completeness of Data Base Sublanguages, in: R.Rustin, (ed.),Courant Computer Science Symposium 6 - Data Base Systems (Prentice-Hall Inc., 1972) 65-98.

[4] Deverell J., The Design of Cellular Arrays for Arithmetic, REE, 1 (1974) 21-26.

[5] Fialkowski K., Muraszkiewicz M., Martix for Retrieval and Sorting, Information Systems, 3 (1980) 219-224.

[6] Flynn M.J., Some Computer Organizations and their Effectivness, IEEE Trans. on Comp., 9 (1972) 948-960.

[7] Franklin M.A. et al., Pin Limitation and Partitioning of VLSI Interconnection Networks, IEEE Trans. on Comp., 11 (1982) 1109-1116.

[8] Kautz W.H., Cellular Logic-in-Memory Arrays, IEEE Trans. on Comp., 8 (1969) 719-727.

[9] Kautz W.H., An Augmented Content-Addressed Memory Array for Implementation with Large-Scale Integration, JACM, 1 (1971) 19-33.

[10] Kung H.T., The Systolic Architectures, Computer, 1 (1982) 37-46.

[11] Maier D., The Theory of Relational Databases (Computer Science Press Inc., 1983).

[12] Mead C., Conway L., Introduction to VLSI Systems (Addison - Wesley, 1980).

[13] Muraszkiewicz M., Concepts of Sorting and Projection in a Cellular Array, Proc. VLDB 7 Conf., (Sept.1981) 76-80.

[14] Muraszkiewicz M., Cellular Networks for Non-numeric Data Processing (in Polish: Sieci komorkowe do przetwarzania danych nienumerycznych), (Prace IINTE no.52, 1984).

[15] Muraszkiewicz M., An Augmented Microcomputer for Relational Database Implementation, in: K.D.Lehmann, H.Strohl-Goebel (eds.), The Application of Microcomputers in Information,

Documentation and Libraries (North-Holland, 1987) 62-70.
[16] Ramamoorthy C.V. et al., A Design of a Fast Cellular Associative Memory for Ordered Retrieval, IEEE Trans. on Comp., 9 (1978) 800-815.
[17] Ullman J.D., Principles of Database Systems (Computer Science Press Inc., 1982).

DLM - A POWERFUL AI COMPUTER FOR EMBEDDED EXPERT SYSTEMS

A. PUDNER BRITISH AEROSPACE

Naval and Electronic Systems Division
Downshire Way
Bracknell
Berkshire RG12 1QL

ABSTRACT: There is an increasing need to operate computers in a real-time environment whereby the solution path of the task cannot be pre-determined. Such problems (eg intelligent process control, vision understanding, data fusion etc) can be solved with an expert system approach that may necessitate interaction with procedural processes. Since expert systems are in general more effectively implemented in declarative languages (eg Lisp, Prolog, Hope), the requirement exists for a powerful AI Computer capable of executing declarative languages and, if necessary, directing the operation of real-time equipment.

The Declarative Language Machine (DLM) is a dedicated operational computer designed and built by British Aerospace for this purpose and incorporates the following key features:

(1) Prolog execution rate of 620 KLIPS (the DLM is believed to be the fastest AI computer in the world).
(2) Direct support for real-time operation and interaction with real-time equipment.
(3) Capability for the efficient execution of Prolog, Parlog, Hope and Lisp.
(4) Capable of being a stand-alone computer or embedded within host equipment.
(5) Floating-point co-processor for arithmetic operations.

This paper describes the architecture of the DLM.

1. INTRODUCTION

1.1. The Requirement For Declarative Languages

Imperative programming languages, for example Fortran and Pascal, are designed for procedural execution on Von Neumann machines. The programmer defines detailed instructions for the computer to follow and every stage of the computation must be described. In addition the detailed behaviour of the computer must be anticipated.

Declarative languages, such as Prolog, Lisp and Hope, are concerned only with relationships that should hold between program input and output (ie the declaration of relationships). Details of how these relationships are utilised in the solving of a problem are left to the machine. Thus whereas for an imperative program the programmer is concerned with both the logic of the problem and the control of that logic, declarative languages require the programmer to be concerned only with the logic.

An imperative program with a specific goal requires the solution path to that goal to be detailed. A declarative program only requires the goal to be specified, and, given the relevant relationships the computer will work out its own solution path.

Problems that are essentially procedural or algorithmic in nature are solved most efficiently using imperative languages. More complex problems, where the solution path cannot be easily defined, or where the solution path needs to adapt to the instantaneous situation, requires more sophisticated programming languages. The complexity of tasks normally associated with intelligence has led to the development of declarative languages as tools for artificial intelligence (AI). For this reason the bulk of serious work in AI is based upon the use of declarative languages.

Declarative languages fall into 2 categories - functional programming languages, e.g. Lisp and Hope, and logic and relational languages eg Prolog and Parlog. Characteristics of the three declarative programming languages Prolog, Parlog and Hope are compared with the procedural language Pascal on Table 1.

1.2 The Requirement for a Powerful Embedded AI Computer

The majority of electronic development work for the civil and defence industries is directed towards compact and cost-effective hardware designed for specific functions. Very few current systems allow for expert system technology to be incorporated. AI techniques could considerably enhance the functional capability of many existing systems and provide a basis for future equipment. It seems likely therefore that many future systems will require the integration of AI techniques into equipment designed for dedicated tasks. Such tasks may include intelligent process control, image understanding, data fusion, and speech recognition etc.

Conventional computers do not execute declarative languages efficiently, and specialised hardware is required. This hardware could take the form of a co-located AI workstation. More desirable however, would be an AI computer specifically designed for embedded real-time applications and with a computational capability significantly greater than any AI workstation available. The BAe DLM is a microcoded computer that provides that hardware solution.

The essential design requirements of the DLM are as follows:-

1) Efficient execution of different declarative languages - both functional and logic. A single language capability would be restrictive.
2) Powerful hardware with a computational capability exceeding the Symbolics 3670 by an order of magnitude.
3) Full real-time operation - the execution of a program must not be halted for system management functions.
4) Hardware to be embedded within host equipment.

The Declarative Language Machine (DLM) has been developed to meet these requirements. Specifically it executes the standard Prolog benchmark program at a rate exceeding 600 KLIPS while requiring only two extended double Eurocards (LIPS, logical inference per

second, are the unit of performance of Prolog programs). The DLM will directly implement declarative language software incorporating Prolog, Hope and Parlog compilers that is being developed at the Imperial College of Science and Technology.

2. DESIGN CONCEPT OF THE DLM

The DLM is not restricted to any specific declarative language. Rather it is hardware implementation of a declarative language virtual machine, called the Σ-Machine (figure 1). The virtual machine instruction set has been designed, by Imperial College, to provide the necessary support for declarative languages. The functional language Hope, and the logic languages Prolog and Parlog are compiled down to the common virtual machine. This approach permits all three languages to co-exist within an AI application thereby allowing the strengths of each to be exploited. A compiler for Prolog exists and Hope and Parlog compilers are planned to be developed. Furthermore an expert system shell (APES) has been developed for Prolog and a comprehensive Prolog programming environment is currently under development, also at Imperial College. This latter project, which is Alvey-sponsored, will provide a program development environment comparable to the most sophisticated Lisp systems.

2.1. Σ-Machine Concept

The Σ-Machine is designed to execute Prolog programs primarily, but also supports functional programming (in particular LISP).

Logic and functional languages have different requirements for their ideal computer, Prolog requiring a more optimised virtual machine than LISP. The instruction set of the Σ- machine has been primarily geared towards Prolog with extra instructions provided to support functional programming. This is valid since Prolog and Lisp share many instructions in common. The instruction set also supports Hope and Parlog by providing facilities for function closures and lazy evaluation.

The memory model of the Σ-machine is typical of many Prolog implementations. The program evaluation requires 4 stacks - the Local Stack (corresponding to the Lisp evaluation stack), the Backtrack Stack, the Reset Stack (or Trail), and the Constructed Term Stack. In the DLM implementation the Local and Backtrack Stacks grow towards each other in the same physical memory.

The Local Stack is used to hold local variables and call activation records. Because of the possibility of a program introducing backtracking choice points, the local stack is not a true stack (it does not always contract when a program completes). This is necessary since environments and call activations are still in use, and may be restarted during future backtracking. The local stack is also contracted during tail recursion optimisation.

In the Σ-machine this optimisation is factored into 2 parts: when entering the last call in a body (corresponding to normal tail recursion in Lisp) and when attempting the last clause in a definition.

The Backtrack Stack contains the activation records for the

various choicepoints introduced during the execution of a Prolog program. It also holds saved arguments which have been passed into Prolog and Lisp programs. This splitting of the argument stack and local stack is unusual for Lisp systems which normally treat arguments and local variables in a single memory area. The advantage in using separate stacks is that the lifetime of an argument is short compared to local variables, and by splitting them, the space used by arguments can be overwritten more quickly than if they shared the local stack.

The Constructed Term Stack contains values which are completed at run-time. The Constructed Term Stack contracts on backtracking but is also garbage collected when it becomes nearly full.

Each memory cell in the DLM evaluation area comprises 38 bits - 32 bits value field, 4 bit tag and 2 bits for garbage collection. These fields can be accessed independently or collectively as required.

2.2. Σ-Machine Instruction Set

The Σ-machine instruction set is made up of 6 distinct categories of instructions:

i) Program call and return instructions.
ii) Clause indexing instructions.
iii) Clause sequencing instructions.
iv) Backtrack control instructions.
v) Unification instructions.
vi) Argument construction instructions.

The instruction set is high level, there being on average one instruction for each symbol in the source code program. A Prolog program consists of a sequence of clauses. The sequencing of alternatives is accomplished by the clause sequencing and the clause indexing instructions. The body of a Prolog rule and Lisp functions are compiled into a sequence of call instructions, one for each of the atoms in the body of a rule. The call instructions build activation records on the local stack and enter the body of the program.

The unification instructions are used primarily by Prolog programs to match the call against some predetermined pattern. The unification instructions are used for structures in the head of a clause. The arguments in the body of a clause are built by the argument construction instructions. These are similar in many ways to the unification instructions, except in that they do not do any comparisons and are therefore simpler.

The backtrack control instructions are used to remove unwanted choicepoints from the backtrack stack thereby removing redundant processing (as defined by the programmer) and reducing the program execution time.

3. DLM ARCHITECTURE

The DLM is a microcoded computer with a hardware architecture dedicated to the support of declarative languages. No system developed to date has combined the essential features necessary for true real-time execution of declarative languages within host

equipment. The basic hardware design concepts of the DLM are discussed in the following sections.

3.1. Sequential Versus Parallel Operation

The fundamental limitations to the speed of execution of any program executed on an "ideal" parallel machine are:-

1) The degree of parallelism that can be exploited from the problem itself - ie the longest chain of events that must be executed sequentially for the task to be completed.
2) The speed of light.

An "ideal" parallel machine generates no overhead for organising the parallel executions of the task on the parallel hardware. For some tasks, eg matrix operations or image processing, the parallelism can be natural and easy to exploit. For the bulk of programs, however, the natural parallelism is limited and its identification is complex. The approach taken in the design of the DLM has been to avoid the complex issues of large-scale parallelism and to concentrate on the development of an essentially sequential system that exploits small-scale parallelism (hidden from the programmer) to provide support for declarative languages. This more conventional approach is less complex and therefore more predictable both in terms of the research and development timescale and the final outcome. It is generally agreed that systems that rely on a radical departure from conventional techniques in both hardware and software usually achieve somewhat less than expected.

3.2 DLM Parallelism

The DLM has been designed to be a direct implementation of the Σ-machine (fig.1). The virtual machine instructions can be viewed as the assembler language of the DLM. Ideally each virtual machine instruction should be executed in 1 hardware clock cycle. Clearly this is a hard figure to achieve. On average, however, a figure of 2-3 clock cycles per virtual machine instruction has been attained for the DLM. This compares with 200-300 clock cycles for a 68000 based computer.

The power of the DLM is a direct result of minimising the number of clock cycles per virtual machine instruction. This minimisation has resulted from the analysis of each virtual machine instruction and the identification of functions that can be executed in parallel. These functions have then been implemented in hardware. The DLM can exploit small-scale parallelism identified within the language execution. The result is that the DLM incorporates 5 processing units and a sophisticated 2-stage program sequencer (fig.2).

3.3. Real-Time Operation

A major problem with declarative languages is the large amount of memory that is used during the execution of a program. Due to the nature of declarative languages much of the data stored at any time is redundant. The problem is alleviated to some extent by partitioning the memory into areas for different types of data with different lifetimes. The DLM has 3 stacks and a heap. The heap contains the majority of redundant data (garbage) and as the limits of the system memory is reached, this area of memory needs

to be reclaimed, only the necessary data being retained and compacted in the base of the memory. This process is known as "garbage collection".

Some well-known systems either explicitly ignore the problem of garbage collection, or, as the limit of the memory is reached, stop the program for several seconds or even minutes, while a garbage collection program reclaims the memory. In the former case, a real-time program will always crash - the question of when being determined by the size of available memory. In the latter case the majority of garbage is generated, and thus the garbage collector is most likely to be called, during a period of maximum activity. In the military situation the period of maximum activity is the time when the system must operate with full capability (i.e. at the onset of an attack or a system failure etc).

The DLM directly supports garbage collection as an essential hardware feature. The existing hardware will garbage collect 1Mcell of memory in less than 1 second. Future designs will aim to reduce this figure to less than 1 TV frame period.

4. DLM HARDWARE

The DLM is a horizontally microcoded computer with a tagged architecture. The micro-code width is 168 bits and is made up of 72 fields. These fields control the operation of the whole system.

The DLM has 6 physical memories that are independently accessable:-

(1) The macro-program store holding virtual machine instructions (64k x 32 bits)
(2) The micro-code store (1k x 168 bits)
(3) The trail store for reset entries (64k x 24 bits)
(4) The main memory area for the stacks and the heap (up to 2M x 38 bits)
(5) The 68020 local memory (64k x 32 bits)
(6) The common memory linking 68020 procedural processor to the declarative language process (64k x 33 bits).

Each main memory cell contains a 4-bit tag field and a 32-bit data field. In addition, 2 extra bits, referred to as "Mark" and "Shift" are used for garbage collection making a total memory cell size of 38 bits. Further hardware support is provided for garbage collection and automatic memory dereferencing.

During any machine cycle up to 5 processing elements operate in parallel:

(1) Data ALU processing the data field.
(2) Tag processor operating on the tag field.
(3) Address ALU performing complex address calculations.
(4) Reset Processor performing reset (or "trail") entries.
(5) An embedded 68020 and 68881 floating-point coprocessor performing algorithmic calculations, I/O and test functions as required.

In addition a 2-stage sequencer simultaneously calculates both the next virtual machine instruction and the next micro-

instruction.

The DLM hardware comprises of 2 hyper-extended double Eurocard boards - the Declarative Language processor (DLP) board and the Intelligent Memory board (see Fig. 3). The DLM uses a combination of ECL, Bipolar and CMOS technologies. Execution of an AI program (which may be developed elsewhere) requires the program to be compiled and downloaded into the macro-store under the control of the embedded 68020 microprocessor. At various stages of the computation data may be passed into and out of the system via the embedded processor. This processor may also execute standard imperative language applications in parallel with AI applications. The co-existance of both declarative amd imperative languages within the same hardware unit is thus provided for.

5. DLM PERFORMANCE FIGURES

Performance measurements have to date concentrated on the commonly used Prolog benchmarks. The DLM executes the Naive Reverse benchmark at 570 KLIP's and the Deterministic Concatonate benchmark at 620 KLIP's. Comparison is made with standard AI work stations on Table 2. Table 2 also includes the parameters of American DARPA-funded PLM and the Japanese fifth generation sequential computer referred to as the HPM. These figures place the DLM performance a factor of two ahead of its' nearest rival. The DLM is compared further with the PLM (executing compiled WAM code) and the IBM PC-AT (executing compiled Σ-machine code) on Table 3. All benchmarks are based on the definition of a logical inference as the successful matching of a clause head. In order to ensure that the comparisons are valid it has been necessary to adjust the published PLM figures to the same definition of a logical inference.

6. CONCLUSION

The design basis of the DLM has been the direct implementation of a virtual machine, the Σ-machine, developed by Imperial College for a range of declarative languages. Rather than being language-specific as is the Warren Abstract machine (WAM), which is highly tuned for Prolog, the Σ-machine is more general and hence useful for AI applications that require a multi-language capability.

The architecture of the DLM has been raised on an analysis of its virtual machine. This has resulted in a micro-programmed computer with "fat" cycle exploiting small-scale parallelism. This compares with the shorter "lean" cycles normally associated with RISC machines. The result is a computer with 5 dedicated processing elements, 6 independently accessable memories, a sophisticated 2-stage sequencer, and the optimum degree of pipelining for declarative languages. The DLM is thus highly optimised for AI processing and at 620 KLIP for the standard Prolog benchmark, it is believed to be the fastest AI computer in the world.

Specific hardware support has been provided for real-time operation in terms of hardware support for garbage collection, a very fast I/O capability and an embedded procedural processor (MC68020) with a tightly-coupled floating-point coprocessor (MC68881). The DLM thus allows for the simultaneous coexistence of both imperative and declarative programming.

7. FUTURE DEVELOPMENTS

The DLM architecture has been refined resulting in the design of a second computer which will be operational during Q4 1987. This machine, referred to as the DLM1, has been included on Table 2 for completeness. The main differences compared to the DLM are:

(i) Further streamlining of the architecture.
(ii) Further intelligence incorporated into the memory.
(iii) Low cost DRAM is used for the main evaluation memory.
(iv) Increased (800 KLIP) performance anticipated.
(v) Capable fo running large (10,000 rule) expert systems.

Research is also continuing into the design of a garbage collectiong algorithm that will garbage collect 2 Mcell of memory in less than 1 TV frame period.

The longer term aim is to develop an ASIC-based design for embedding in AI system.

8. REFERENCES

1. CLARK K. and McCABE F. "Micro-Prolog: Programming in Logic" Prentice - Hall International 1984.

2. DOBRY T.P. et al - "Design Decisions Influencing the Microarchitecture for a Prolog Machine". Proc IEEE 1984 - CH2088

3. DOBRYT.P. et al - "Performance Studies of a Prolog Machine Architecture". Proc. IEEE 1985 - 0149-7111

4. GABRIEL J. et al - "A tutorial on the Warren Abstract Machine" Argonne National Laboratory Technical Report.

5. McCABE F. - "Σ-machine Specification 1986.

6. McCABE F. - "Logic and Objects". Imperial College Report DOC 86/9.

7. MILLS J.W. - "A High Performance LOW RISC Machine for Logic Programming".

8. NAKAZAKI R. et al - "Design of a High-Speed Prolog Machine (HPM)". Proc. IEE 1985-0149-7111

9. TAMURU N. et al "Sequential Prolog Machine PEK" Proc. Int. Conf on FGCS 1984

10. TICK E. "Sequential Prolog Machine: Image and Host Architecture". Proc. IEEE 1984 - CH 2088

11. TICK E. "An Overlapped Prolog Processor" SRI Technical Note 308.

12. WARREN D.H.D. - "An Abstract Prolog Instruction Set". SRI Technical Note 309 October 1983.

A.R.G. PUDNER British Aerospace NESD, Downshire Way
Bracknell, Berkshire RG11 1QL
Tel: 0344 483222 Ext. 8740

Declarative Language Machine (DLM) Concept:

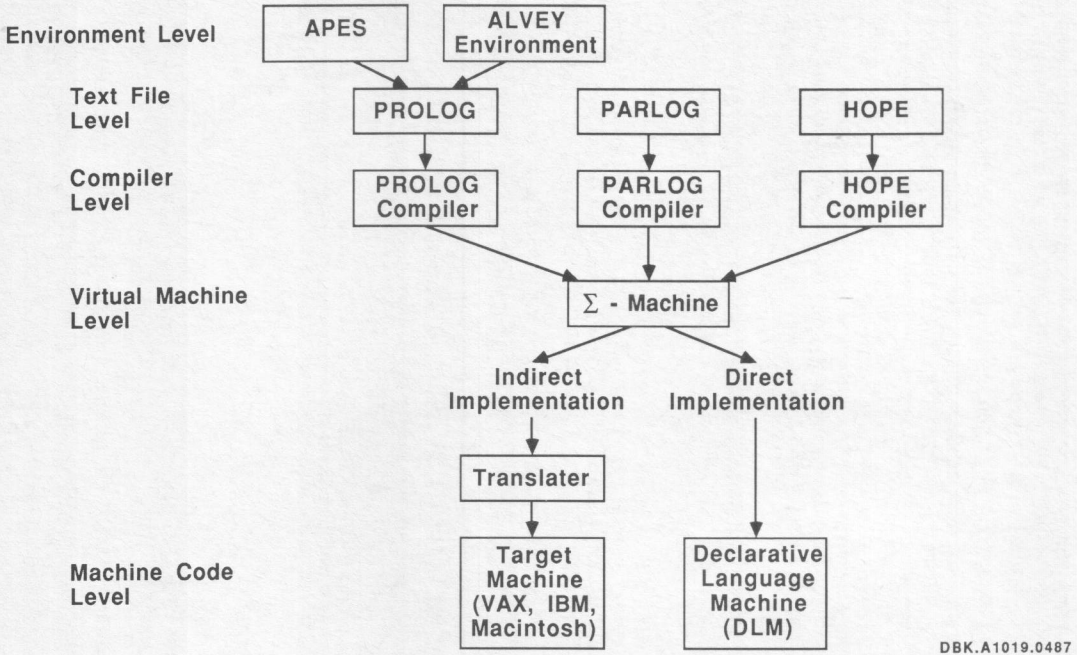


Figure 1

DLM Architecture

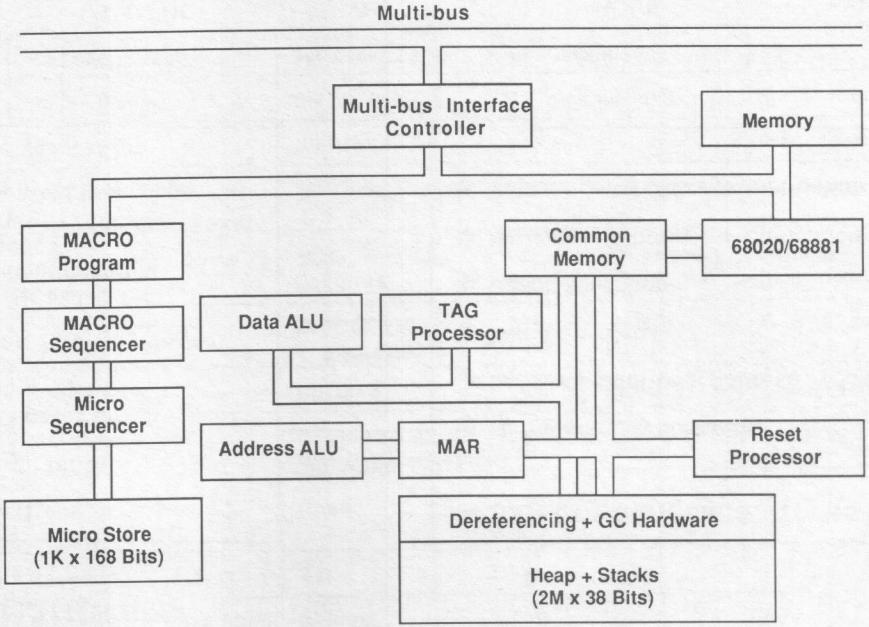
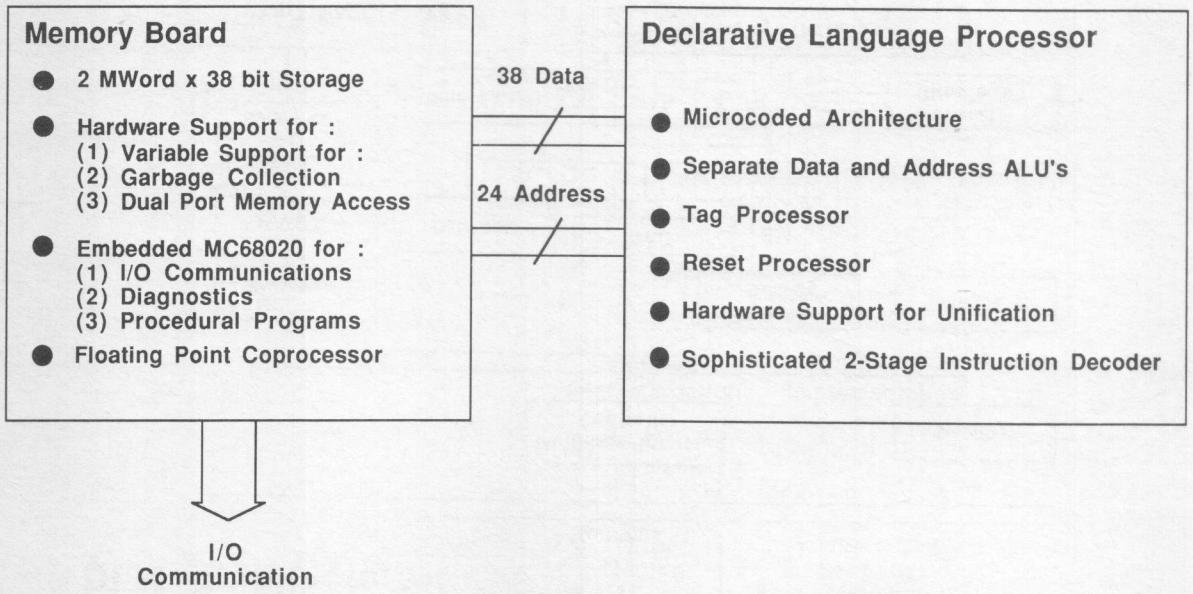


Figure 2

DLM Hardware



DBK.A1015.0487

Figure 3

Comparison of Languages

Parameter	PROLOG	PARLOG	HOPE	PASCAL
Formal Basis	1st Order Logic	1st Order Logic	λ - Calculus	B.N.F.
Program Structure	Set of Clauses	Set of Clauses	Set of Equations	Set of Procedures
Program Evaluation	Unification	Unification	Reduction	Procedural
Program Order	1st Order	1st Order	Higher Order (Functions of functions)	1st Order
Variable Directionability	Variables Multimode	Mode must be Declared	Variables Directional (Specified I/P or O/P)	Variables Directional (Specified I/P & O/P)
Variable Expressiveness	Logical Variables	Logical Variables	Ground Variables	Ground Variables
Typing	Untyped	Untyped	Strongly Typed	Strongly Typed
Determinism	Non-Deterministic	Committed Choice Non-Deterministic	Deterministic	Deterministic
Program Transformation from Specification	Good	Good	Very Good	Average
Parallel Evaluation	Restricted	Yes	Yes	No
Compiled Execution Speed	Medium (Same as LISP)	Medium	Fast	Fast

Table 1

DBK.A1017.0487

BRITISH AEROSPACE

Comparison of AI Computers

Parameter	Orion	SUN 3	Symbolics 3675	HPM (ICOT FGCS)	PLM (Berkeley)	DLM (BAe)	DLM1 (BAe) Q4 1987
Architecture	GP	GP	Dedicated	Dedicated	Dedicated	Dedicated	Dedicated
Parallelism	No	No	Small-scale (3 Processing Elements)	Small-scale (4 Processing Elements)	Small-scale (2 Processing Elements)	Small-scale (5 Processing Elements)	Small-scale (5 Processing Elements)
Memory Speed	150 nsec	150 nsec	150 nsec	100 nsec	?	35 nsec	100nsec
Memory Width	32 Bits	32 Bits	36 Bits	36 Bits	32 Bits	38 Bits	38 Bits
Technology	Bipolar	Bipolar/ CMOS	Bipolar	CML	Bipolar	Bipolar	CMOS + Bipolar
μ - Code Width	64	N/A	112	80	144	168	200
Virtual Machine	Σ - Machine	WAM	WAM	WAM	WAM	Σ - Machine	Σ - Machine
Prolog Execution Rate	20 KLIP	50 KLIP	70 KLIP	270 KLIP	229 KLIP	620 KLIP	800 KLIP Expected
Cycles for Append Benchmark	~ 500	~320	~100	35	~40	15	9

Table 2

DBK.A1148.0987

DLM Performance Comparison (KLIP's)

Benchmark	IBM PC-AT	PLM	DLM
Deterministic Concatenate (Con 1)	8	229	620
Naive Reverse	7.9	115	570
Non-deterministic Concatenate (Con 6)	3.9	84	256
Quickstart	2.6	91	203
Naive Reverse in reverse	0.7	---	265

Table 3

DBK.A1334.0887

WAVE-1: A NEW IDEOLOGY OF PARALLEL AND DISTRIBUTED PROCESSING ON GRAPHS AND NETWORKS

Peter S.SAPATY

V.M.Glushkov Institute of Cybernetics,Ukranian Academy of Sciences, Glushkov Avenue 20, 252207 Kiev, USSR

A new ideology as well as the corresponding language WAVE-1 are presented for highly parallel and distributed processing on large amounts of tightly bound and complex structured data. They are based on parallel propagation in the united space & processing continuum, formed by active data networks, of the special recursively defined wave formulas. These formulas, spreading through the active data on multiple wavefronts, may cause arbitrary complex actions on data including creation, modification or deletion of the passed subnetworks. The proposed approach may be efficiently used in designing the new generation high-performance knowledge base machines as well as for organizing the distributed solution of complex tasks in computer networks, representing distributed information or control systems.

1. INTRODUCTION

The main cause of inefficiency of the traditional computers for processing complex graph and network data structures lies in the ruling model of computation, based on the activity of a program (having usually a limited number of simultaneously active branches) which operates on a huge amount of physically unbound and passive data. An essential step forward has been done by distributing the processing power among the data, attaching it, as local activities, to the data items in vertices of the graph or network data structures, which may do local processing and exchanges with the neighbouring vertices [1-8].

This analog-like approach to parallel and distributed processing in large network knowledge bases may result in the level of parallelism proportional to the volume of knowledge bases, when supported by the special hardware and software environment.

The main works on computers and languages based on active network data structures, however, concern the SIMD-type ideology of parallel processing where the same instruction is broadcast to all active vertices of the net [4,5]. The whole program running these associatively executed instructions remains sequential, so such parallel machines may not process in parallel all the task at once, but only small and homogeneous parts of it, thus considerably restraining their potential power for solving complex graph and network problems.

In [9,10] a new approach has been proposed where the program in a special high-level wave language does not reside in any control

unit but is propagating, as the multiple wavefronts, through the active data network in a highly split and replicated manner. This permits to solve arbitrary complex tasks in active data in an extremely concurrent and distributed manner, without the traditional control. In [10] the WAVE-0 language was proposed which is based on the new, recursively defined, parallel navigation structure over the data networks, in exchange for the conventional control structures of usual languages. Being only an amalgamation of the wave navigation structure with a number of special functions, embracing different parts of it, the WAVE-0 proved to be efficient for parallel solutions of complex retrieval and inference tasks on semantic networks. Adding the proper interpretations to this WAVE-0 skeleton, it is possible to obtain new type, wave, languages for the distributed solving of different classes of tasks on large amounts of tightly bound and complex structured data.In the present paper we briefly describe the WAVE-1 language, the next one of the wave type, which presents some universal tool for the parallel and distributed solving of different classes of tasks on general graphs and networks.

2. THE WAVE MODEL OF DISTRIBUTED PROCESSING

2.1. The Data Network

In this paper we consider solving different classes of graph and network problems on some generalized data model, called the Data Network (DN) which is defined as follows. DN (some example of it is shown in Fig.1) consists of vertices and arcs, oriented as well as not oriented, connecting these vertices. Each vertex is associated with data item which may correspond either to a name of some concept or to arbitrary value (being called further a name or a value of the vertex). The name of the vertex is represented by an identifier whereas the value is set by arbitrary string of characters in quotation marks. Each arc has a name set by identifier, and vertices, in general, may have more than one arc with the same name coming into them or emanating from them. In such a case we shall say that some arc name identifies a multiarc coming into the vertex, emanating from it or incident to it, when both cases take place (depending also on the orientation we attach to the name). Two vertices may be connected by arbitrary number of arbitrary oriented but differently named arcs.

In the DN there may be more than one vertex having the same data item,so, pointing some name or value of the vertex in the DN we identify, in general, some set of vertices, say, multivertex, in it. The originality of the elementary vertices or arcs may be established, taking into account their contexts in the DN, in general. The values assigned to the vertices of DN may represent the most different kinds of information: numerical or textual data elements, data structures, bodies of procedures and so on. The fact that vertices may have either names or values (but not both) does not restrict the generality of DN model because, for instance, for the case where the vertex named A must have some value named B whose meaning is ´C´ we may form the vertex identified by the value ´C´which is conncted to vertex A by an arc named B.

2.2. The Active Data Network

The described DN model for its distributed processing is then converted into the special parallel machine-oriented model which

we shall call the Active Data Network (ADN). The ADN (an example of which, corresponding to the DN in Fig.1, is shown in Fig.2) consists of nodes and links which are defined as follows. For each vertex of the initial DN model there exists a corresponding node in the ADN having its unique personal address and keeping,besides

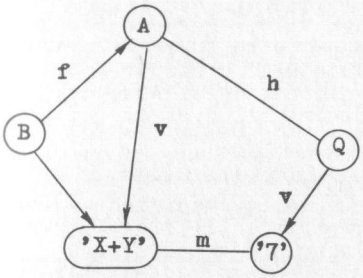


FIGURE 1

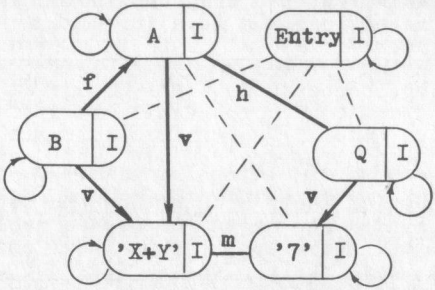


FIGURE 2

the data item from the DN, a special activity I, enabling it to do local processing on this item, storing the intermediate results, exchaning messages with the other nodes and handling a number of local control functions necessary for distributed processing.There are next kinds of links between nodes of ADN: surface links, corresponding to the arcs of the initial DN (and having their names), extra links and loops. The surface links (shown in Fig.2 in hard) represent bidirectional channels permitting to pass messages both along and opposite the orientation of the corresponding arcs.Their names and orientation are pointed together with the addresses of the nodes from which they emanate or to which they lead in both nodes connected by them. Each surface link has its personal order in relation to each node it is incident to.

The extra links (some of which are shown in Fig.2 by dotted lines) enable to transfer messages from each node of the ADN directly to each other one using special kinds of internode communication. The transference of messages through both surface and extra links may be either selective (to the dedicated nodes) or broadcasting (to all the nodes these links lead to). The loops are used as some abstraction for description of virtual propagation of wavefronts in data space when they physically remain in the same nodes of ADN. All nodes of ADN may access a number of common channels binding ADN with the external world which may be used concurrently by different nodes. It is also assumed that all nodes of ADN may work concurrently and asynchronously, exchanging messages with other ones, without any centralized control device. The special entry node (or a number of them) is added to ADN having only extra links with other nodes of ADN.

2.3. The Wave Organization of ADN Behaviour

The activity I in each node of ADN is provided by the same copy of the interpreter of the special wave language in which all tasks to be solved by ADN are presented. The main feature of this language is that programs in it operate not over the data structure, as usual, being constantly allocated to one or to a number of processors operating on common or distributed data, but among the active

distributed data, spreading throught them and dynamically forming directly in their space integrated with the data parallel operational structures. The latter may work in a highly distributed both data processing and control mode and combine efficiently the two well-known modes of parallel processing: associative and dataflow. The programs in the wave language in the form of recursively defined wave formulas are put initially into the entry node of ADN and begin to spread through it by the node activities using surface and extra links, and loops, with their selective or broadcasting modes of operation. This wave propagation may take place concurrently in different directions of ADN, creating, in general, multiple wavefronts proceeding all over the active data structure.

During their propagation the wave formulas having the special syntax and semantics may be replicated, split, modified by the node activity, loosing their utilized parts until finally exhausting their possibilities for colonizing and transforming the data space. Propagating in space, the wave formulas may invoke distributed processing of data, extracting or synthesizing copies of the subnetworks covered by them. They may do pattern matching of one structures represented by wave formulas with the others that are contained in a distributed form in the ADN, as well as creation, reconfiguration or deletion of the ADN substructures, and so on. The wavefronts, carrying the wave formulas, may transfer together with them different kinds of information obtained in some parts of ADN to other parts of it. The described wave model integrating the distributed data with the network of intelligent interpreters of the special wave language, being supported by the special parallel architectures, may provide a very high performance of the new generation knowledge base machines which may be proportional, in general, to the volume of knowledge bases.

3. THE WAVE-1 LANGUAGE

3.1. General Description of the Language

A program in WAVE-1 language is represented by a wave formula (or, merely, wave) which is initially applied to the entry node of ADN and concurrently propagates through it. The syntax of WAVE-1 is as follows.

```
wave            = front { . front}
front           = sector { , sector}
sector          = {key} [ replicator * ] generator
generator       = propagation | action | ⟨wave⟩|
                  [function] (wave [ ; extension ]) | call
propagation     = polyarc [ #polyvertex]
polyarc         = multiarc | ( multiarc { , multiarc} )|variable
multiarc        = orientation bond | empty
orientation     = + | - | ~ | empty
bond            = identifier | ○|⊗|order | ? [variable]
polyvertex      = multivertex | (multivertex { , multivertex})
                  variable
multivertex     = identifier | 'string' | address
call            = CL( [ parameters ] ; source ) | variable
function        = grasping | reconfiguration | raking
grasping        = ES | EP | IS | IP | AS | AP
reconfiguration = MS | SS | DS
```

```
raking              = RS | RE | RN
variable            = @{letter_or_digit} | !
extension           = declaration | source | destination | range
declaration         = type : variables
type                = T | S | P
key                 = % | ∇ | ↓ | ↑ | ⊥
range               = limit [ : limit]
parameters = variables      = variable { , variable}
replicator = limit          = natural | ∞ | variable
source = destination        = channel | variable
address = channel = order   = natural
```

In the syntax description [A] means that A may be omitted, {A} - that A may be repeated zero or more times, "=" is used for definition of syntactic categories, and "|" is a delimiter of alternatives.

As may be seen, at the upper level a wave is represented as a sequence $f_1.f_2. \ldots .f_i. \ldots .f_n$ of some composite constructs f_i called fronts, which are separated by a dot. Each f_i describes a current step of parallel wavefront propagation in ADN, carrying the tail $f_{i+1}. \ldots .f_n$ of the wave. The process of the wavefront propagation terminates when the tail of the wave becomes empty. Each f_i may consist of a set of sectors $s_1, s_2, \ldots, s_i, \ldots, s_m$ (with the sectors separated by comma), which reflect the peculiarities of wavefront propagations in different physical or virtual directions in ADN. Each sector, in general, is a sequence $k_j r_j * g_j$ where k_j is a set of keys detailing operations on wavefront propagation, g_j is a generator - the main construct causing concrete wavefront movement, and r_j with a delimiter "*" establishes a number of applications of g_j. So $r_j * g_j$ is equivalent to $(g_j.g_j. \ldots .g_j)$ with g_j repeated r_j times. g_j may be simple or complex. A simple g_j may correspond either to elementary wavefront propagation or to elementary action defined over the system of special space variables and causing the wavefront circulation throught loops. A complex g_j may correspond to arbitrary wave structures in brackets thus setting recursive definition of the wave formulas, resulting in arbitrary complexity of the wave navigation and data processing procedures. A wave in round brackets may be also an argument of the space functions embedded into the language, which attach different kinds of control and service to the wavefront propagation caused by subformulas they embrace. An elementary propagation starting at the current vertex (i.e. the one belonging to the set of vertices in which the wavefront currently resides) is generally represented by a couple polyarc#polyvertex delimited by "#". The polyarc identifies some or all elementary arcs incident to the current vertex that are to be passed and the polyvertex (if pointed) allows them to come only to definite types of vertices. Multiarcs and multivertices from which the polyconstructs are composed correspond, in general, to sets of equally identified elements.

Multiarc may be set by indicating the type of bond between vertices prefixed by its orientation ("+" or empty for direct passing the arcs, "-" for reverse and "~" for arbitrary direction or for undirected bonds). The bond is mostly pointed out by identifier or order. Special characters "$\bigcirc$" and "$\otimes$" define two special types of associative multiarcs, where the former causes the wavefront propagation to all other vertices of DN (or, correspondingly, nodes of ADN) and the latter - only to all neighbouring

vertices (or nodes of ADN). "○" corresponds to the extra links of the ADN model. These multiarcs give very powerful means for organizing both local and global associative processes in arbitrary vertices of ADN. An empty multiarc names loop returning the wavefront to the same vertex. Multivertex may be set by identifier, by arbitrary set of characters or by physical address in ADN, generally corresponding in the first two cases to sets of equally defined vertices. Generators, polyarcs and polyvertices may be also pointed out by names of space variables whose meanings are dynamically inserted into the wave formula during the wavefront propagation. Generators, moreover, may be also injected into the wave structure as procedures with parameters by the Call (CL) construct, the bodies of procedures being either accessed as meanings of variables or input from the outside through the common channels. The very parts of the wave subformulas may be present in ADN as values of vertices, being assigned as meanings of the proper variables by the same wavefront, if needed.

During the wave propagation there may appear distributed in ADN the special space variables that may belong to the following classes: transitive (T), stationary (S) and personal (P), the duration of their existence in ADN corresponds to the time of propagation of the wave subformulas containing their declarations. The T- and S-type variables appear in vertices which find their declarations in the current wave subformulas. The T-type variables propagate on the wavefronts serving the passed vertices whereas the S-type variables remain static in places of their birth, being shared by the vertices passed by the corresponding wave formula. The P-type variables appear in every vertex where access to them took place, serving only these vertices. The meanings of the variables accessible from the current vertex are formally considered as temporary vertices in ADN connected to the current one by arcs named by the variable names and prefixed by "?" ("?" alone names a temporary multiarc leading simultaneously to the meanings of all variables accessible from the current one). The T-type variables migrate with the wavefronts and their meanings become bound to the vertices keeping the wavefronts. The space variables are represented in the wave by special identifiers beginning with "@". "!" names a special variable corresponding to the item of the current vertex. "∞" is used for pointing some upper bound or infinitely large value.

The elementary actions that may take place in vertices, passed by wavefronts, may be either filters, describing conditions having to be true to pass the loop, or assignments, preceding looping. Both filters and assignments are defined over the variables and may in the simplest case have the following syntax:

```
action      = filter | assignment
filter      = element compare element
assignment  = variable [ ← [element ] ] operator element
compare     = = | ≠ | < | ≤ | > | ≥
element     = integer | identifier | 'string' | variable
operator    = + | - | × | / | &
```

where " & " is used for string concatenation, " ← " means assignment, different forms of assignment are possible (including implicit). Self-identification of data and automatic type conversions are implied.

The vertices reached by a wavefront, starting in some vertex v

and propagating according to the front f_i, may be conditionally subdivided into the following three classes. A Set of Examined Vertices (SEV) includes all vertices reached according to f_i. A Set of Active Vertices (SAV) in the simplest case, where the wavefronts propagate only to a one-arc depth, corresponds to SEV, but in general it equals to the SAV obtained by the last front from which f_i is composed, assuming that SAV on different sectors are united (i.e. being defined recursively). The vertices of SAV on f_i behave afterwards as new sources for wavefront propagation caused by the next front f_{i+1} or are used in different operations produced by functions. A Set of Passed Vertices (SPV) on f_i includes all the descendants of v from the SEV on f_i, lying on the passed paths from v to vertices of the SAV on f_i. The construct $\langle w \rangle$ makes all vertices from the SPV on w active just outside the angle brackets, i.e. SAV on $\langle w \rangle$ corresponds to SPV on w.

3.2. The Space Functions and Keys

There are 12 main standard space functions embedded into the language which add special kinds of distributed control and service to the basic mechanisms of wavefront propagation. They are primarily activated in vertices discovering the functional symbols at the heads of the formulas passing through them, which behave then as leaders of distributed execution of functions in the ADN structure. After the termination of processes on colonization and transformation of ADN by subformulas enclosed by functions and passing the tails of the waves to SAV produced by functions, the leaders lose their guiding property. The space functions are divided into three main classes: grasping, reconfiguration and raking.

The grasping functions first split the wave formula they embrace into a set of branches where such a splitting takes place on the first front of subformula and the tail of the subformula becomes the tail of each branch. The first front is split either statically, when it may be explicitly decomposed into a set of sectors, or dynamically, on the SAV produced by it, each vertex of this SAV being an entry to the separate branch. The grasping functions initially select some subset of this set of branches, called a clipping, and then initiate parallel or sequential multicoloured wavefront propagation by different branches, each branch having its personal colour. According to the peculiarities of concrete functions, the SAV obtained by different branches are amalgamated. The resulting SAV on grasping functions is established as a subset of this amalgamation, whose vertices have been crossed only by a permissible number of distinctly coloured wavefronts, produced by different branches. The range of this number as well as the parameters for setting the clipping may be either explicitly pointed in the extensions of wave formulas, or implied.

ES (Exclusive or Sequential) and EP (Exclusive or Parallel) functions select only one branch from the clipping that gives non-empty SAV. ES activates the wavefront propagation sequentially on branches and stops its work when the current branch gives non-empty SAV, whereas EP activates all branches concurrently and selects the first one in time giving non-empty SAV, killing activity of the others. IS (Inclusive or Sequential) and IP (Inclusive or Parallel), correspondingly, sequentially or in parallel activate all branches from the clipping and amalgamate their SAV (some branches may give empty SAV). AS (And Sequential) and AP (And

Parallel) working in sequential or parallel mode amalgamate SAV on all branches only in case SAV on each branch from the clipping is not empty. Obtaining an empty SAV on a current branch, AS stops its work. So does AP if even though one of the branches, working in parallel, gives an empty SAV. The grasping functions, covering by control the proper regions of ADN, selecting and compressing the results, may be regarded as integral space interpretations of many important mechanisms from conventional languages: switches, indexing, conditionals, parallel control and synchronization, set operations (of both classical and fuzzy modes), etc., working in a highly parallel and distributed mode.

The reconfiguration functions establish a special mode of wave processes in ADN where the wave formulas, spreading on wavefronts, may change the very ADN. MS (Modify Structure) is the most general function of this class which permits , using the wave syntax, to establish arbitrary changes in ADN, considering the wave it embraces as a system of distributed productions. SS (Supplement and Substitute) and DS (Delete Structure) make only partial changes where the former may extend ADN or substitute the data items in nodes and the latter deletes the parts of ADN passed by the wavefront. SS and DS are used in cases where the wave formula they embrace was obtained by raking functions and is used to modify the ADN.

The raking functions, activated in leaders, permit to collect in them information about the subnetworks that are colonized by the corresponding subformulas. RS (Raking Structure) function makes a copy of the colonized subnetwork as a bound structure in the wave syntax. RE (Raking Elements) collects information as sets of elements (arcs or vertices), and RN (Raking Number) presents the results as a number of elements passed by the wavefront. All information obtained by the raking functions concerns only the SAV on the corresponding wave subformula. The information about the colonized part of the ADN being raked up together in the leaders, may be subsequently assigned as meanings to the space variables and used in the wave processes or output beyond the ADN through the common channels.

The keys are used to modify or detail a number of operations in the wave processes. " % " is used only inside the angle brackets. It masks the activity of vertices obtained by the wave it embraces, i.e. it erases from SAV, obtained by the wave in angle brackets, some subset. " ∇ " permits to reenter vertices by wavefronts with the same colours (without this key such vertices become terminal for these wavefronts). " $\downarrow$ " is used with raking functions and substitutes the elements in extracted copies by their addresses or orders. For instance, inclusion into the obtained wave formula (by RS function) of physical addresses of vertices gives powerful means for its further application in the wave processes with some fulcra outlined. " $\uparrow$ " in the context of DS function causes deletion of only arcs in ADN structure, and in the context of RS function it masks in the obtained wave formulas the information about vertices, thus generalizing this wave as a template for subsequent wave processes. " $\perp$ " presents an additional means for linearization in the wave syntax of complex graph templates: the subformula it embraces, is regarded as a terminal one in the linear, string,wave representation.

4. QUESTIONS OF DISTRIBUTED INTERPRETATION OF THE LANGUAGE

The described WAVE-1 language and the wave ideology are oriented from the very beginning on their application in the new generation high-performance knowledge base machines organized as multiprocessor structures with distributed memory and control. The interpretation of the WAVE-1 language in such structures will consist in distributed navigation of the wave formulas together with retaining tracks in the multiprocessor space during the wavefront propagation. These are used for distributed execution of the space functions, garbage deletion and system control.

The distributed navigation of wave formulas involves the next main operations, which may be carried out independently and in distributed manner in different nodes of ADN: selecting the head of the wave formula, splitting it into sectors, replicating the common tail and concatenating it to each sector, forming independent waves, passing the tail of each wave through all arcs whose names are pointed in the heads of the waves (if in the selecting and splitting processes the simple generators are reached) and so on. Space functions, keys and space variables demand additional control of the wavefront propagation, which may be also efficiently distirbuted in ADN.

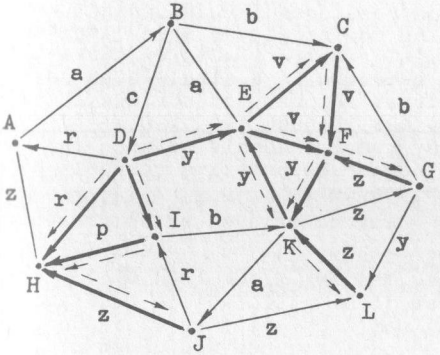


FIGURE 3

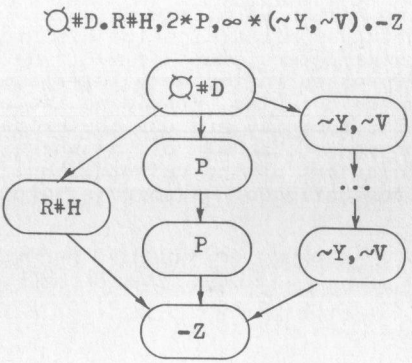


FIGURE 4

In Fig.3 some DN is illustrated through which a wave formula, representing some unrigid template (shown in Fig.4 together with the graph of its structure), is propagating and colonizing the subnet reflected by hard arcs. The dotted arrows point at the directions of the wavefront propagation.

The most evident may be a direct support of the ADN model by reconfigurable networks of microprocessors where each processor keeps one node of ADN (i.e. a data item and a copy of the wave language interpreter). The inefficiency of such direct support of ADN at present state of computer techniques may be conditioned by difficulties of on-line rearrangements of connections among a large number of processors and memory expenses to keep the copy of rather sophisticated interpreter per each data item. More attractive nowadays seems the placement of ADN into the high-organized regular multiprocessor structures [11] where the surface

links of ADN are mapped onto the regular local interconnections among the processors (possibly, with transits through neighbours) and the extra links as well as different kinds of system control are supported by special additional hierarchy of communication means among processors, enabling the direct transmissions of messages between arbitrary couples of processors.

Each processor may contain in its local memory, in general, some subnetwork of the DN where a part of arcs of DN, allocated in the memory of the same processor, is passed by the wavefronts virtually, whereas the physical transitions of them take place only when the arcs connect vertices situated in different processors. With small personal memories of processors it may be reasonable to organize processors in clusters sharing the common memory blocks keeping the whole interpreter. Such clustering may be hierarchical where, on higher levels, these memory blocks may correspond to distributed devices of external memory keeping also non-resident parts of DN. Designing the special microprocessors for direct hardware support of wave languages as well as supporting the extra links of ADN by new physical principles (optical, laser, electromagnetic fields, etc.) may be of particular interest.

The proposed wave ideology may be rather efficient also in organizing distributed solution of tasks in computer networks representing distributed knowledge base or control systems and consisting of the mainframe or special-purpose computers. Such networks, being supplied with the distributed copies of the wave language interpreters, may solve in parallel arbitrary complex tasks expressed in the wave formalism, behaving as a single high-performance computer. Providing the natural support for surface links of ADN, they may process the extra links through the special telephone or radio channels or, at worst, by usual channels between the computers of the network, emulating the associative requests by propagation of wavefronts through the net.

5. EXAMPLES OF THE WAVE PROGRAMS

The proposed wave formalism proved to be efficient in describing parallel distributed solutions of different classes of tasks on graphs and networks. Some examples of them are shown below.

5.1. Building in the active space a semantic network with arbitrary structure. We shall consider two possible solutions for the net shown in Fig.5.

(i) ∇SS(◯#A.↓(RE(;@A).-O#B.(RE(;@B).-L#C.-N#@A,
(J#D.I#@A),(-K#E.-M#@B);T:@B);T:@A)).

In this case the net creation starts from vertex A and then a spanning tree of the net is unfolded in distributed manner (illustrated in Fig.5 by arcs shown in bold). The two transitive variables @A and @B are used for fixation the physical addresses of the obtained A and B vertices, which are further used for proper distributed attaching of the remaining arcs, forming contours.

(ii) SS(◯#C.(↓AS((J#D.I#A.RE(;@A)),-N#@A,
(L#B.RE(;@B).O#@A),(-K#E.-M#@B));S:@A,@B)).

Here the net is unfolded from vertex C using two stationary variables.

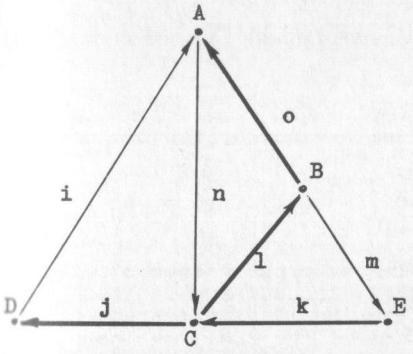


FIGURE 5

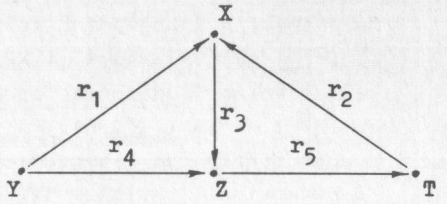


FIGURE 6

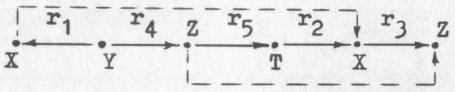


FIGURE 7

5.2. Finding the critical path of the net diagram. Suppose such a diagram (without cycles) has a single initial vertex A, single terminal vertex F, and each arc is marked by the duration d of some operation (being positive). As only vertices in the DN model may have values assigned to them, each arc d is substituted in DN by a fragment ⟶ d ⟶ with a vertex, having d as a meaning, and two arcs with some arbitrary name r. A modification of the distributed algorithm from [8] is used where the wavefront appears at the initial vertex A and propagates concurrently through the tree evolvent of the net diagram, terminating in vertices in which the meaning of the earliest time of the event may not be changed by the current wavefront. For the simplicity, the critical path is output in the reverse order.

```
(AS((◯#A.∇∞*(↓RE(;@).R.@D+!.R.@D≻ET.@ET←@D.
@PR←@;T:@,@D),RS(⟨◯#F.∞*◯#(@PR⟩));P:@ET,@PR)
```

Here the transit variables @ and @D are used, correspondingly, to pass a vertex address and a meaning of the time of event to the successor vertex. The personal variables @ET and @PR keep, correspondingly, the meaning of the earliest time of the event and the address of the predecessor vertex that established this time. Initial zero values for variables are implied. Not pointed destination for RS function means output to some standard channel. " ∞ " identifies here arbitrary large number of repetitions of the corresponding subformula.

5.3. Distributed pattern matching on a semantic network. The wave ideology and syntax of the wave formulas present a wide variety of possibilities for solving tasks, related to pattern matching, in parallel and distributed mode on general graphs and networks. In Fig.6 some graph template is shown. Let us consider some examples on matching this template with arbitrary semantic network, represented by ADN, and determining the meanings of the free variables that correspond to the vertices of the template, assuming, for generality, that all its vertices are unknown.

(1) Determining only one free variable (a set of its possible meanings) when the template in the wave syntax starts to unfold from some another variable (i.e. from the vertices that may

correspond to it).

- Determine Z beginning from X:
  IP(⌬.AP((-R1.R4),R3,(-R2.-R5);∞).RE()) .
- Determine T beginning from X:
  IP(⌬.AP((AP((-R1.R4),R3;∞).R5),-R2;∞).RE()) .
  (Similarly for Y beginning from X, and for Y and T from Z.)
- Determine T beginning from Y:
  IP(⌬.AS((R4.@Z←1),(R1.AP(-R2,(R3.@Z=1.R5);∞).RE()));P:@Z).
  (Similarly for Y beginning from T.)

The sets of meanings of variables are gathered in some standard channel, their subsets being previously accumulated in leaders, guiding the RE function. All wave formulas illustrated above begin to unfold concurrently in all vertices of the ADN, "∞" here points the maximally possible intersection of vertices by differently coloured waves, the number of colours being equal here to the number of branches into which the formulas are split.

(2) Finding all possible combinations of all meanings of the free variables X,Y,Z,T for the template in Fig.6, using the wave formula based on the linear form [6] of that template, shown in Fig.7, which reflects some path through all its vertices:

```
∇IP(⌬.(RE(;@X).-R1.(RE(;@Y).R4.(RE(;@Z).R5.(RE(;@T).
  R2#@X.R3#@Z.RS(?);T: @T);T:@Z);T:@Y);T: @X)) .
```

(3) Display all possible solutions for all variables in the form of the united tree-like structure without explicit pointing all their separate combinations, assuming the template is unfolded from vertex X

```
∇RS( ⟨ IP(⌬.AP((-R1.R4),R3,(-R2.-R5);∞))⟩) .
```

The unity in the wave formula of both procedural and declarative knowledge, being combined with powerful space functions and space variables, enables to represent arbitrary graph patterns in the wave syntax that may be of exact and fuzzy, unrigid, character. The latter opens a new way to the parallel and distributed solution of complex problems related not only to isomorphism but also to similarity of graph and network structures.

At present the WAVE-1 us used to describe distributed solutions of classical tasks on graphs and networks, problems related to expert and CAD systems, calculations and sorting on networks, information retrieval and parallel inference on semantic networks. In the latter case the set of clauses, representing some logical task, is converted into the set of graphs, which is subdivided into the assertion graph, containing only ground terms, and the goal structure, including free variables [6,7]. The latter, being presented in the wave syntax, is injected into the active assertion graph existing as the ADN. The meanings of free variables may be found in a matching process, similar to the previously described examples. The experimental library of parallel wave programs in WAVE-1 language contains now more than 100 programs from the very different classes.

6. THE WAVE MACHINE ARCHITECTURE

The overall organization of the concrete version of the high-performance wave machine directly interpreting the wave language, which is currently under investigation, is shown in Fig.8. It is

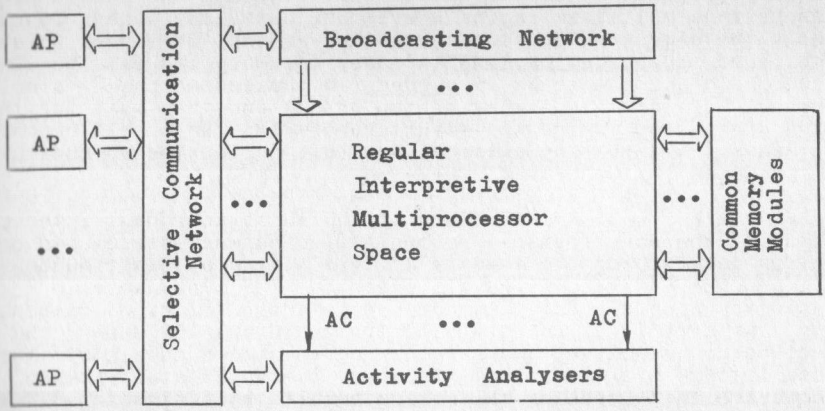


FIGURE 8

based on a Large Regular Interpretive Multiprocessor Space (RIMS) with local interconnections among the processors having a limited number of bonds with the other processors. Each processor has its own local memory and may also access the distributed in RIMS Common Memory Modules (CMM) of large volumes, serving processors in their vicinity. The processors may communicate with each other either directly, through the local connections, or by sending messages through the other ones.

Communication through the local connections within the RIMS represents only one of the three possible types of interprocessor communication means. The two others may be realized by the Selective Communication Network (SCN), enabling direct communications between every two components of the machine shown in Fig.8 (including all processors of the RIMS), and by the Broadcasting Network (BN) permitting to pass the same messages to all processors of the RIMS (or to some subsets of them) from the Access Processors (AP) or from arbitrary processors from the very RIMS.

The BN may be invoked into operation through the SCN for which it is considered as one (or a number) of abonents. The BN sending in parallel the same messages to a large number of processors of RIMS and concurrently from a number of sources, may have hierarchical structure with pipeline and asynchronous modes of operation.

The ADN model is mapped onto the RIMS by the locality principle where the surface bonds among the nodes of ADN are close to the local interconnections among the processors. Depending on the real volume of the knowledge base and the number of processors in RIMS, each processor may keep, in general, a subnetwork of the ADN model. Each processor also keeps a resident part of the wave language interpreter, whereas the full texts of it are contained in the CMM, as well as non-resident parts of the ADN. The necessary

non-resident parts of the interpreter are dynamically loaded into the processors of RIMS during the wave propagation through the ADN and parsing the wave formulas in different processors of the multiprocessor space.

The wave programs enter the wave machine through the access processors in a multitask regime where many users may access concurrently the same knowledge base existing as the ADN model, which is directly supported by RIMS. AP pass the wave programs to RIMS through SCN using also BN if needed (in the second case - simultaneously to many processors of the space) and then wait for the results obtained inside the structure of RIMS in a self-organizing manner, without any external control, during the spreading of the waves through RIMS.

One of the key problems of the distributed asynchronous interpretation of the wave formulas in the active network-structured data lies in determining the moments of termination of distributed wave processes taking place all over the RIMS. The two main strategies are used for this purpose in their efficient combination. One of them is in retaining tracks in the ADN during the wavefront propagations and sending the special echo signals through these tracks, starting at points where the waves stop. These echoes are then used for collecting results in a structured form, garbage deletion from the ADN after interpretation of the wave formulas as well as for checking the overall termination of the wave processes (using "and" logic for their merging in the last case).

Another efficient strategy, permitting not to control in details every path of the wave propagation and often having a much less overhead, is in listening the activity of the wave processes distributed in a great volume of the environment by using the special signal Analog Channels (AC) and the appropriate Activity Analysers (AA). In this case certian waves may be assigned personal AC (from some common pool), signalling about their activity during their propagation in ADN. The appropriate AA listening a noise in these channels check the appearance or, vice versa, disappearance of certain activities in ADN. The invokation of these listening machanisms may be done from the access processors as well as from arbitrary processors from RIMS (in the second case some parts of the multiprocessor space may totally control the activity of the other parts).

The processors of RIMS as well as the AP are oriented on string processing and have special hardware mechanisms for handling the dynamic and virtual memory and producing the analog activity signals on different frequencies through a number of common AC.

In Fig.9 a possible organization of RIMS is illustrated which is based on the transputer-like modules having four outcomes. Three of these outcomes are used for communications with the other modules forming a regular space closed by the outlying modules. The remaining, fourth, outcome may be connected to one of the following resources of the wave machine: SCN, BN or one of the local common memory modules CMM_i evenly distributed throughout the RIMS. Bonds with SCN and BN^i are also evenly dispersed in RIMS. So, every elementary module of RIMS may communicate with SCN, BN or CMM either directly or through one of its neighbouring modules. The fourth outcome of the modules may be also used for

communication with AA by sending digital activity signals, however the supplying modules of RIMS with the special analog devices for emitting signals into common AC may be much more efficient.

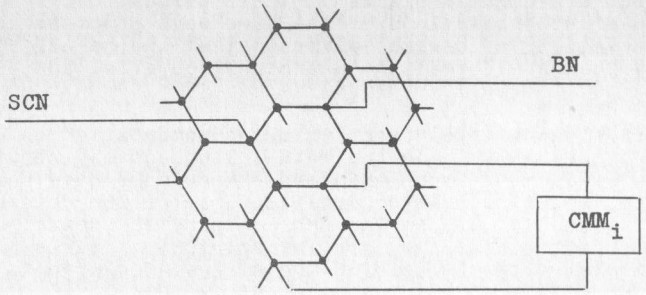


FIGURE 9

The proposed organization of a high-performance knowledge base machine,directly supporting the ADN model, has different kinds and levels of communications among the processors which may be in many cases mutually replacable. Most data exchanges take place by the local connections of the RIMS, possibly using transits through the other processors. When, however, a sender and a receiver happen to be situated too far in the space, or the traffic through the local connections of RIMS becomes faulty or overloaded, the messages may be passed through SCN. The broadcasting of messages by BN to many processors may be emulated by spreading of parallel wavefronts through RIMS, or done sequentially by SCN. Listening the activity of wave processes through the analog channels may be modelled by sending digital messages from many points of RIMS to certain processors, using both local interconnections and SCN (of course, in a far less efficient manner), and so on. Such communication heterogeneity and redundancy may considerably increase the reliability of the wave machine architecture together with permitting the efficient specialization of different forms of interprocessor communications, caused by distributed interpretation of the wave languages, spreading through the active data networks.

CONCLUSIONS

We have presented a completely new formalism of distributed solving tasks in large amounts of tightly bound and complex structured data. It consists in propagation of the very program body on the multiple wavefronts in the united space & processing continuum, formed by the active network-structured data. The wave formulas written in the WAVE-1 language, being some universal tool for arbitrary complex distributed creation, modification and processing of graph and network data structures, may be looked at from different points of view. Fisrt, they may represent graph templates expressed in a special syntax, which, when being applied to the graph or network data structures, represented by the ADN, find in a matching process meanings of the necessary variables. Second, they are regarded as creative models on a high level of abstraction that dynamically form, in the context of certain data and their interconnections,

concrete high-parallel operational structures working directly in data space in the unity of associative and dataflow modes of operation. Third, they may be thought of as special expressions of controlled production systems where the sets of productions are spreading across the ADN and finding data to which they may be applied, directly in places of their allocation. The variety of possible interpretations of the same wave programs results also in a variety of classes of tasks that may be efficiently solved by the wave formalism in a highly parallel and distributed manner.

The described wave model of distributed computation on complex data structures as well as the WAVE-1 language may result in creating new classes of high-performance knowledge base computers for solving tasks related to artificial intelligence and working in both distributed processing and distributed control modes, as well as new methods of the parallel solution of tasks in distributed computer networks, giving the performance proportional to the volume of data.

This model as well as WAVE-0 and WAVE-1 languages constitute at present a basis of the project PAMIR concerning the creation of high-performance knowledge base machines. This project is being developed at the International Basic Laboratory (BL) for Artificial Intelligence with the Institute of Technical Cybernetics of the Slovak Academy of Sciences in Bratislava, Czechoslovakia, in cooperation with the Computer Centre of the USSR Academy of Sciences in Moscow and V.M.Glushkov Institute of Cybernetics of the Ukranian Academy of Sciences in Kiev, USSR. The WAVE-0 language (being a subset of the WAVE-1) has also become a kernel of the Information System of the BL where the data about countries, topics, organizations, time, persons, etc. are amalgamated as an integral heterogeneous semantic network over which the powerful wave search techniques is realized [12]. This system is now operational on the IBM PC in a Lisp environment. The works on the WAVE-1 language currently concern a development of basic methods of its efficient parallel and distributed interpretation on a highly organized multiprocessor environment directly supporting the ADN model where Lisp and OCCAM languages are being used for the simulation and comparison of different solutions.

The presented wave model of highly parallel and distributed processing on general graphs and networks which does not seperate data, operations on data and interpretation of the special wave languages on these data may permit to considerably alleviate many traditional bottlenecks of the other approaches, for example, caused by the information traffic between the inference and data base engines in the Japanese fifth generation project [13], by the separated control in the SIMD-like projects [4,5] , or by the potential difficulties of processing the intricately structured data within the dataflow organization[14,15] where the bound data structures must be split into sets of separate tokens propagating through the network of operations.

ACKNOWLEDGEMENTS

Many thanks to D.A.Pospelov (Computer Centre of the USSR Academy of Sciences) who inspired the formation of this philosophy, to W.D.Hillis (Thinking Machines Corporation) and L.Bic (University of California, Irvine) for their support of this work during the author's stay at the Basic AI Laboratory in Bratislava, as well as to R.Vollmar (Technical University, Braunschweig) and D.Parkinson (DAP Support Unit) for very useful discussions of the ideas of this paper. The author is also grateful to the anonymous referees whose comments helped much in preparation of the final version of this paper.

REFERENCES

[1] Finkel,R.A., Solomon,M., Horowitz,M.L., Distributed Algorithms for Global Structuring, National Computer Conference, AFIPS Conference Proceedings, Vol.48, AFIPS Press, Montvale, New Jersey, June 1979.

[2] Chang,E.J.H., Echo algorithms: Depth Parallel Operations on General Graphs, IEEE Trans. Software Engineering, SE-8,4, 1982.

[3] Chandy,K.M., Misra,J., Distributed Computation on Graphs: Shortest Path Algorithms, Comm. ACM, Vol.25, No.11, November 1982.

[4] Hillis,W.D., The Connection Machine, The MIT Press Series in Artificial Intelligence, Ph.D.Thesis, MIT,1985.

[5] Moldovan,D.I., Tung Y.-W., SNAP: A VLSI Architecture for Artificial Intelligence Processing, J. of Parallel and Distributed Computing, Vol.2,No.2,1985.

[6] Bic,L., Execution of Logic Programs on a Dataflow Architecture, Eleventh Ann. Int´l Symp. Computer Architecture, Ann Arbor, 1984.

[7] Bic,L., A Data-Driven Model for Parallel Interpretation of Logic Programs, International Conference on Fifth Generation Computer Systems, Tokyo, Japan (1984).

[8] Sapaty,P.S., Active Informational Field as a Model of Structural Solving Tasks on Graphs and Networks, Proc. USSR Academy of Sciences. Technical Cybernetics, 5,1984.

[9] Sapaty,P.S., A Wave Approach to the Languages for Semantic Network Processing, Proc. Int´l Conference on Knowledge Representation. Section 1: Artificial Intelligence, Kiev,1984.

[10] Sapaty,P.S., A Wave language for Parallel Processing of Semantic Networks, Computers and Artificial Intelligence, Vol.5, No.4, 1986, Bratislava, Czechoslovakia.

[11] Sapaty,P.S., Kočiš,I., A Parallel Network Wave Machine, Proc. III International Workshop PARCELLA´86, Akademie-Verlag, Berlin 1986.

[12] Sapaty,P.S., Varbanov,S., and Dimitrova ,M., Information Systems Based on Wave Navigation Techniques and Their Implementation on Parallel Computers, Proc. Working Conference "Knowledge and Vision Processing Systems", November 24-28, Smolenice, Czechoslovakia,1986.

[13] Aiso,H., Fifth Generation Computer Architecture, Proc. Int´l Conf. Fifth Generation Computer Systems (Oct.,81).

[14] Dennis,J.B., First Version of a Dataflow Procedure Language , In Lecture Notes in Computer Science, Vol.19,G.Goos and J.Hartmanis, Eds.,Springer, New York.

[15] Veen,A.H., Dataflow Machine Architecture, ACM Computing Surveys, Vol.18, No.4, December 1986.

BRAVE - A PARALLEL LOGIC LANGUAGE FOR ARTIFICIAL INTELLIGENCE

T.J.Reynolds, A.J.Beaumont, A.S.K.Cheng,
S.A.Delgado-Rannauro and L.A.Spacek

University of Essex
Wivenhoe Park
Colchester CO4 3SQ
UK
e/mail JANET reynt@sx.cscvax

This paper presents the first results for the implementation of the logic language BRAVE on a parallel architecture. We explain the operational semantics of BRAVE with common programming examples and show how both AND and OR parallelism can be exploited and controlled using BRAVE syntax. The design of an abstract machine for the parallel execution of BRAVE is given along with the principles of compilation and example codings. Results are presented from running example programs on a three-processor prototype, using an interpreter for BRAVE written in C.

1. INTRODUCTION

BRAVE is a logic language which allows a programmer to exploit both AND and OR parallelism as most suits the application. We include OR-parallelism because we believe that many Artificial Intelligence problems exhibit large search spaces suitable for OR-parallel exploration. Examples abound in planning, scene interpretation and parsing natural language. In particular we have translated WARPLAN into BRAVE (Kefalas 1987), and investigated worlds that are complex enough to contain many alternative actions at each point in the formation of a plan. An OR-parallel search of these worlds is advantageous, both in terms of finding any plan, and particularily in finding the optimal one.

Our approach is, in this respect, completely different from that of the

committed-choice non-deterministic languages, PARLOG (Clark and Gregory 1986), Concurrent Prolog (Shapiro 1983) and GHC (Ueda 1985). In these languages the alternatives for matching to a particular goal are protected by guards. As soon as one of these guards succeeds, then execution is committed to that clause and it is not possible to simultaneously investigate alternative choices, i.e there is no OR-parallelism. This has the implementational advantage that only one global binding environment need be maintained, but of course only one solution for a given goal is obtained.

Other groups taking an OR-parallel approach to Prolog include those cooperating in the Gigalips project - Manchester University (Shen and Warren 1987, Warren 1987), Argonne National Laboratory USA (Butler et al 1986) and the Swedish Institute for Computer Science (Hausman, Ciepielewski and Haridi 1987). In order to be competitive an implementation of OR-parallelism requires careful management of multiple binding environments. We are employing a scheme based on that of D.S. Warren (Warren 1985), which is very similar to the SRI model independently suggested by D.H.D Warren (Warren 1987).

It may not be desirable to expand all OR-choices in parallel: inherently sequential behaviour may be desired (as for I/O); or it may be necessary to control parallelism to fit the proof into available memory. We provide annotations to allow the programmer to decide where to use OR-parallelism.

2. THE BRAVE LANGUAGE

2.1 AND connectives

There are two forms of AND connective available in BRAVE - ',' and '&'. If subgoals are joined by ',' then solutions (tuples of bindings) are communicated left-to-right as constraints. For the following conjunction :

```
g1 , g2
```

the goal g1 will be activated first, then bindings produced by the first solution will be used to activate g2. While an attempt is made to satisfy g2 another solution will be sought for g1, to be passed in its turn. This is a pipelined parallelism over separate solutions to subgoals.

The conjunction '&' is used to indicate that subgoals should be satisfied in parallel, unconstrained. It is up to the programmer to decide that there are no shared variables, or possible data dependencies between the two sub-goals.

For example :

g1 & g2

indicates that g1 and g2 will be activated in parallel, and as bindings are produced which satisfy the goals independently another process will pass the conjunct solution to the caller.

2.2 OR connectives

Disjoint clauses are connected in BRAVE by ':' to indicate that they should be tried in parallel whilst clauses to be tried sequentially are terminated by '.'. Thus if we write

```
g:- p:
g:- q.
g:- r.
```

attempts will be made to satisfy the goal g by simultaneously trying to satisfy both p and q. All solutions for p and q will be found, and passed to the caller, before r is tried.

2.3 Single solution only

Sometimes it may be desirable that only one solution which proves a particular relation be sought. This is implemented in BRAVE by a built-in predicate :

single(goal)

Using this predicate the first set of bindings that are discovered to satisfy the goal will be committed to. There will be no consideration of alternate bindings in case of subsequent failure of another conjunct goal. This implies that the portion of the proof tree that seeks alternate bindings can be pruned immediately single commits.

2.4 Collecting and expanding lists

Frequently it is desirable to collect all the solutions for a particular relation into a list - a facility provided in Prolog by bagof and setof, and conversely

to allow all members of a list to be examined in turn - usually coded in Prolog by member and by using backtracking. In BRAVE this process is crucial and both jobs are performed by one powerful built-in predicate.

set(arg1, goal, L) — gathers into list L all instances of arg1 (variable or term) that will satisfy the goal ; returns [] if none do.

set(arg1, goal, [...]) — will unify arg1 with each item of the list successively, and then call goal. set will succeed for each instance that goal succeeds and any variables in arg1 will be exported; set will fail if the third argument is instantiated to anything other than a non-empty list.

The well-known member predicate can now be defined in BRAVE as

```
member(A,L):- set(A,true,L).
```

This simply succeeds if both A and L are instantiated and A is a member of list L. If, however, L is instantiated to a list and A is uninstantiated then A will be succesively instantiated to each of the elements of L. This is an efficient coding of member because the recursion of the normal definition has been turned into an iteration at a lower level, within the implementation. Append may be defined using set in both directions:

```
append(L1,L2,L3):- set(X, join(L1,L2,X), L3).
join(L,_,X):- set(X,true,L).
join(_,L,X):- set(X,true,L).
```

The crucial point to realise about this definition is that the two instances of join are an **ordered** disjunction. When bindings for X are collected in the body of append to form list L3, the elements of L1 are collected first, then the elements of L2.

A useful further control over disjunctions is provided by '!' . Clauses in a disjunction after this terminator are tried only if the previous clause has failed. This is useful for terminating conditions and for protecting special cases. For example in

```
delete(A,[],[])!
delete(A,L1,L2):- set(X, set(X, X=A, L1), L2).
```

The second instance of delete will be tried only if the first has failed.

The next example program determines if two binary trees are isomorphic and

is taken from Codish and Shapiro (1986). It illustrates the utility of having both OR and AND parallelism in maintaining a clear style. Two trees are isomorphic if their subtrees are isomorphic in any order and two leaves are isomorphic if they are identical.

```
isotree(leaf(X),leaf(X)):
isotree(tree(A,B),tree(C,D)):-
        isotree(A,C) & isotree(B,D):
isotree(tree(A,B),tree(C,D)):-
        isotree(A,D) & isotree(B,C).
```

2.5 Passing results as they are produced

It is highly desirable to pass solutions for a relation to the caller as they are produced. This allows the programming of 'monitors', perpetual processes which communicate via solutions as they are passed, as well as providing solutions immediately when perhaps only the first few are required.

3. BRAVE ABSTRACT MACHINE ARCHITECTURE

The operational semantics of BRAVE is based on independent processes communicating by means of the logic variable. We base our abstract machine on the Warren Machine (D.H.D.Warren 1983), with the addition of instructions for the creation of processes, process communication and synchronisation. Scheduling from a list of process descriptors is used to map processes onto physical processors. Spawning more processes than processors has a distinct advantage in architectures where communication latency is high, as processors may switch context, rather than wait. Each process will inherit an environment of variable bindings from its parent, accessed by a pointer aregs which allows indexing into a set of a-registers. A process will construct an environment, using b-registers for its children to inherit. Thus the abstract machine run-time structure will consist of a process tree, off which hang the environments, which in turn point to actual data structures.

Each node in the process tree is represented by a process descriptor (PD) with the following fields:

pc	Pointer to next instruction to be executed
cp	Continuation pointer to more code
pcp	Previous continuation pointer (points to a PD)
aregs	Pointer to A registers

bregs	Pointer to B registers
fl	Pointer to forward list
report	Pointer to compose result location

3.1 Control Flow

A conventional stack onto which return addresses are pushed is not appropriate to this machine. A process may simply die (if cp and pcp are nil), reporting its termination, or pick up a continuation found from either cp or pcp. If there are more subgoals within the clause body to execute then these will be pointed to by cp. If there is further clause body to execute higher in the tree, then this will be found by following pcp to an ancestor PD which has a non-nil cp.

4. PRINCIPLES OF COMPILATION

We now show how the different parallel and sequential program connectives map into process control instructions.

4.1 Disjunctions

At the beginning of the code for clauses of a given predicate there will be a sequence of switch_on_reg instructions forming a decision tree compiled to test the arguments in the head of the call. The terminals for the decision tree will be the OR-cases for a given clause compiled into a chain of create_process with a type (sequential, parallel or conditional) determined by the terminator. For example the program

```
h(1) :-b1.
h(2) :-b2:
h(3) :-b3:
```

compiles into

```
h/1:    switch_on_reg b0    VAR:[CL1,CL2,CL3];1:[CL1];2:[CL2];3:[CL3]
CL1:     create_process sequential
         execute b1/0
CL2:     create_process parallel
         execute b2/0
```

```
CL3:    create_process parallel
        execute b3/0
```

For the goal h(X) all clauses must be tried. The parent will create a process to find all results for b1, and processes for b2 and b3 will be spawned in parallel when all the results for b1 have been received.

If the first terminator is (!) then the code is similar, but the type of the create_process is conditional. Processes for b2 and b3 will only be created if b1 fails to produce any results.

4.2 Conjunctions

In order to handle the conjunction ',' a continuation pointer to the right-most goal is given to the left-most goal, which is activated by 'call'. This ensures that all OR-cases will continue to prove the conjunction. For example the clause

```
h :- d, e, f.
```

compiles into

```
h/1:    create_process   sequential
        call  d/0              cp = L1
L1      call   e/0             cp = L2
L2      execute   f/0
```

Notice that call is used to activate both d and e. This is so that all processes that are generated by the call to d (the or-cases for d) will have the continuation pointer cp pointing to L1, where they will continue on termination. The same will apply for e, but there is no continuation for f, as the body is finished.

4.3 Managing alternate binding environments

In sequential backtracking Prolog at most one binding for each variable exists at a time. In order to implement OR-parallel search for multiple solutions, we need a method of storing and accessing efficiently many possible bindings for each variable. We are employing a method where each active leaf of the search tree creates a 'forward list' of the bindings made to those variables which may be bound by more than one process. When a process expands a particular path it caches the bindings found from searches through forward

lists into a hash table. The key for the hash address being the identifier for the variable. To look up a binding for a particular variable, we look first to its address. If it is unbound there, we look into the hash table to find out if a binding exists for that variable. We are currently examining different algorithms for the most efficient updating of the hash table when a processor switches to expand a different path.

5. RESULTS

An interpreter for BRAVE has been coded in C to run on a SUN-3. Process creation and scheduling is simulated in this sequential version. Here is a comparison of results obtained from running BRAVE, PARLOG, and C-PROLOG versions of some simple programs.

RUN-TIMES ON A SUN-3 RELATIVE TO C-PROLOG			
PROGRAM	C-PROLOG	BRAVE	PARLOG
nreverse(50)	1	6.01	6.48
qsort(50)	1	6.23	8.07
scene analysis	1	3.03	-
4queens	1	3.53	-

Our interpreter has not been optimised in any way and it is likely that the overhead of OR-parallel execution indicated by these results could be reduced by at least a factor of 2. We feel that we have sufficient encouragement to proceed with the compiler where it is reasonable to expect that the overheads will be less.

In order to get some feel for the parallelism available in example programs we have modified the BRAVE interpreter to simulate parallel execution whilst running on a single processor SUN-3. This simulation works by visiting each processor in turn to perform a resolution, if there is any work to do, before visiting the next processor. One cycle is defined to be complete when we have visited each processor once. The number of cycles taken to return all solutions becomes a measure of the runtime of any program. By adding more processors we can see how the number of cycles decreases for any particular program.

Figure 1 shows the graph of the speed-up gained by adding more processors to the four queens problem. We can see that by adding 30 processors we can obtain a speed up factor of about 18. The scene labelling and family

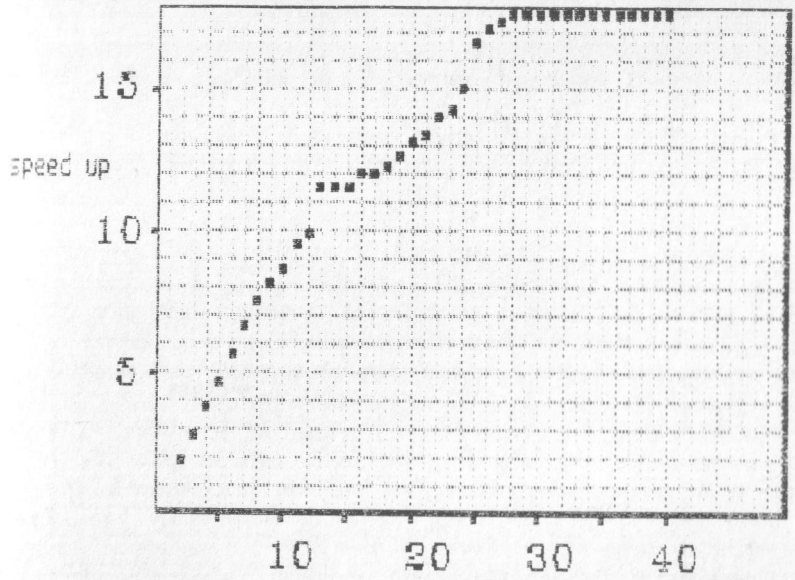

15
speed up
10
5
10
20
30
40
number of processors

Fig 1

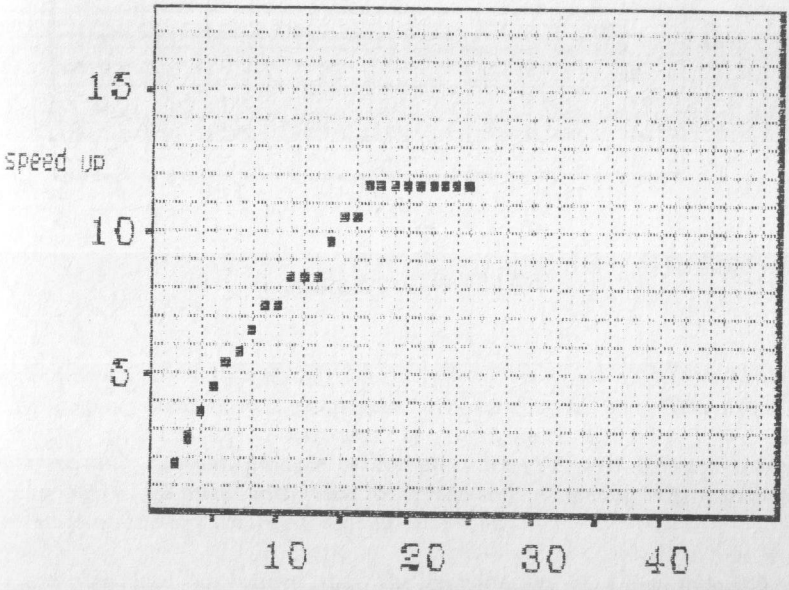

15
speed up
10
5
10
20
30
40
number of processors

Fig 2

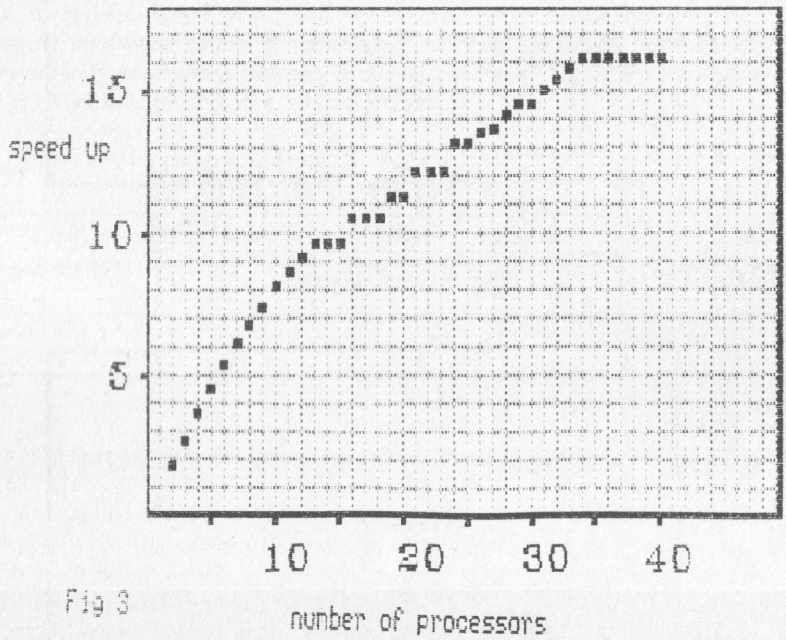


Fig 3

database programs both show respectable available parallelism (figures 2 and 3).

There are obvious problems in interpreting the results from any simulation, and in particular, it is easy to be over-optimistic. In order to get some feeling for the overheads of task switching, involving possible re-construction of hash tables, we have also tried out our interpreter on a real parallel machine.

This prototype machine consists of three processor cards in a VME backplane, sharing access to a global memory. The processing element consists of a MC68020 and local memory. In the local memory the interpreter and the compiled program are stored, whilst the global memory stores the run time data structures, the process descriptors and the runnable process list.

In the next results, we can see that deterministic programs exhibit no speed-up. We can also see that the speed up is not linear for programs that do exhibit parallelism. There are two main reasons for this. Firstly, processing elements may spend some time idle when executing programs that exhibit insufficient parallelism. Secondly, a sequential processor can cache the binding environment effectively, whilst multiple processors will sometimes be forced to switch context, rendering this optimisation less effective.

EXECUTION OF BRAVE ON THE ESSEX PROTOTYPE			
#processors	1	2	3
nreverse(50)	1.00	1.01	1.00
qsort(50)	1.00	0.99	0.99
scene analyis	1.00	1.71	2.30
4queens	1.00	1.52	2.08

6. CONCLUSION

There is every reason to suppose that the benefits of compilation will be the same for parallel logic languages as for sequential ones. We therefore anticipate that the performance of compiled BRAVE will be such that BRAVE running on a machine with tens of processing elements will perform better than state-of-the-art compiled sequential Prologs.

We are also lead to believe that the exploitation of OR-parallelism in logic languages for fifth generation architectures is the right approach for the support of applications in Artificial Intelligence.

ACKNOWLEDGEMENT

This research is supported by the Science and Engineering Research Council of Great Britain through Alvey IKBS Grant 135. Sergio Delgado is supported by the Mexican Government through grant 44860.

REFERENCES

R. Butler, E.L. Lusk, R. Olson and R. Overbeek, ANLWAM: A Parallel Implementation of the Warren Abstract Machine. Internal Report, Argonne National Laboratory, USA 1986.

K.L. Clark and S. Gregory, PARLOG: Parallel Programming in Logic. ACM Transactions on Programming Languages and Systems Vol 8, No 1. Jan 1986.

M. Codish and E. Shapiro, Compiling OR-parallelism into AND-parallelism. Third International Conference on Logic Programming, London July 1986.

S. Gregory, Sequential Parlog Machine Specification. Internal Report, Imperial College London 1985.

B. Hausman, A. Ciepielewski and S. Haridi, OR-parallel Prolog Made Efficient on Shared Memory Multiprocessors. Research Report SICS 87006, Swedish Institute of Computer Science, 1987.

P. Kefalas, WARPLAN - A Planner in BRAVE. MSc Dissertation, University of Essex Sept 1987.

T.J. Reynolds, A.J. Beaumont and L.A. Spacek, The Design and Implementation of an AND/OR parallel Logic Language. University of Essex Internal Report CSM-92, Nov 1986.

E. Shapiro, A Subset of Concurrent Prolog and its Interpreter. ICOT Technical Report TR-003, Feb 1983.

Kish Shen and David H.D. Warren, A Simulation Study of the Argonne Model for Or-Parallel Execution of Prolog. IEEE Symposium on Logic Programming, San Francisco Sept 1987.

K. Ueda, Guarded Horn Clauses. ICOT Technical Report TR-103, June 1985.

D.H.D. Warren, An Abstract Prolog Instruction Set. SRI International Technical Note 309, Menlo Park CA 1983.

D.H.D. Warren, The SRI Model for OR-parallel Execution of Prolog - Abstract Design and Implementation Issues. IEEE Symposium on Logic Programming San Francisco Sept 1987.

D.S. Warren, Efficient Prolog memory management for flexible control strategies. In Proc. Int. Symp. on Logic Programming pp 198-202 IEEE Atlantic City N.J. Feb 1984.

APPENDIX A - BENCHMARK PROGRAMS

```
/* naive reverse */

append([],L,L)!
append([A|X],Y,[A|T]):- append(X,Y,T).

reverse([],[])!
reverse([X|Y],L):- reverse(Y,L1), append(L1,[X],L).

/* 4 queens */

queens(N,Qs):-
            range(1,N,Ns),
            queens(Ns,[],Qs).

queens(UnplacedQs, SafeQs, Qs):-
            select(Q,UnplacedQs,UnplacedQs1),
            not(attack(Q,SafeQs)),
            queens(UnplacedQs1,[Q|SafeQs],Qs).
queens([],Qs,Qs).

select(X,[X|Xs],Xs):
select(X,[Y|Ys],[Y|Zs]):- select(X,Ys,Zs):

attack(X,Xs):-
            attack(X,1,Xs).

attack(X,N,[Y|Ys]):-
            X is Y+N!
attack(X,N,[Y|Ys]):-
            X is Y-N!
```

```
attack(X,N,[Y|Ys]):-
        N1 is N+1,attack(X,N1,Ys)!

range(M,N,[M|Ns]):-
        M<N,
        M1 is M+1,
        range(M1,N,Ns)!
range(N,N,[N])!

/* scene analysis */

l(V,od(_,V),od(V,_)):
l(V,od(V,_),od(_,V)):
l(V,od(V,_),convex):
l(V,convex,od(_,V)):
l(V,concave,od(V,_)):
l(V,od(_,V),concave):

y(_,convex,convex,convex):
y(_,concave,concave,concave):
y(V,od(_,V),od(V,_),concave):
y(V,concave,od(_,V),od(V,_)):
y(V,od(V,_),concave,od(_,V)):

arrow(V,convex,od(_,V),od(V,_)):
arrow(_,convex,concave,concave):
arrow(_,concave,convex,convex):

scene(qube, [A,B,C,D,E,F,G,H,I]):-
        arrow(a,G,F,A),l(b,A,B),arrow(c,H,B,C),
        l(d,C,D),arrow(e,I,D,E),l(f,E,F),
        y(g,G,H,I).
```

APPENDIX B - EXAMPLE CODING FOR QUICKSORT

```
qsort([],[])!
qsort([U|X],Z):-
        partition(U,X,X1,X2),
        qsort(X1,Z1), qsort(X2,Z2),
        append(Z1,[U|Z2],Z).

qsort/2:
        switch_on_reg  b0  [VAR:CL1,[]:CL4,LIST:CL5]
```

```
CL1:    switch_on_reg  b1  [VAR:CL2,[]:CL3,DEFAULT:CL5]
CL2:    try_all_clauses  [CL4,CL5]
CL3:    try_all_clauses  [CL4,CL5]
CL4:    create_process  conditional  2
        unify_nil  b(0)
        unify_nil  b(1)
        proceed
CL5:    create_process  sequential  12
        unify_list  b(4)  b(5)  b(0)
        unify_variable  b(6)  b(1)
        put_value  b(4)  b(0)
        put_value  b(5)  b(1)
        put_variable  b(7)  b(2)
        put_variable  b(3)  b(3)
        bcall  partition/4
        put_value  b(7)  b(0)
        put_variable  b(8)  b(1)
        bcall  qsort/2
        put_value  b(3)  b(0)
        put_variable  b(9)  b(1)
        bcall  qsort/2
        put_value  b(8)  b(0)
        put_value  b(9)  b(10)
        put_value  b(4)  b(11)
        put_list  b(11)  b(10)  b(1)
        put_value  b(6)  b(2)
        bcall  append/3
        proceed
```

This code is as it is produced by our current compiler and certain obvious optimisations have not yet been incorporated. In particular re-use of b registers is often possible.

STUDY OF MECHANISMS THAT SUPPORT THE IMPLEMENTATION OF NON-LOGICAL COMPONENTS OF PROLOG IN WAM-BASED SYSTEMS

Zhang Chenxi and Tzu Yungui

Department of Computer Science
Changsha Institute of Technology
Hunan, P.R. of China

Most of the recent efficient Prolog systems are based on the WAM (Warren Abstract Machine) [1]. But the original WAM provides little support for the implementation of non-logical components of Prolog. In this paper, an execution model for non-logical components proposed by the authors is presented. The ideas of modifiable code and operation modes for DBOP-predicates are proposed in the paper. The semantics of modifiable procedure code and considerations on the implementations of code databases and source databases are discussed.

1. INTRODUCTION

Although Prolog is a programming language based on logic, it contains non-logical components, i.e., built-in predicates, which play an important role in enhancing the power of Prolog language and in improving the execution efficiency of Prolog programs. Among them, the most fundamental ones are DataBase OPeration predicates (which we call DBOP-predicates) which provide the programmers with the ability to modify the databases at runtime. They are of great significance in many applications of Prolog, e.g., knowledge management systems. And we believe that they are necessities in Prolog systems of pratical applications.

The performance of compiler-based Prolog systems greatly outstrips that of interpreter-based systems. Serious Prolog implementations in recent years have been primarily compiler-based [2], of which most are based on the WAM. The Warren Abstract Instruction Set is a high performance Prolog-oriented object code. It has the properties of being suitable for implementation both in software (either interpreted or compiled into native code) and in hardware or firmware.

But new problems arise when compilation techniques are used. It is much more difficult to implement DBOP-predicates in compiler-based systems than in interpreter-based systems due to the fact that the structures of code are quite different from that of data. The original WAM is only an execution model for logical components of Prolog. It provides little support for the implementation of non-logical components. We have extended the WAM, and proposed a model for the implementation of non-logical components, which is described in the paper. The building blocks of the model are first discussed in Section 2. Then schemes for implementing code databases, source databases and the DBOP-predicates are discussed in Section 4, 5, and 6 respectively.

2. BUILDING BLOCKS OF THE EXECUTION MECHANISMS OF NON-LOGICAL COMPONENTS

In a compiler-based system, clauses comprising Prolog programs are stored in the form of sequences of executable instructions, which is quite different from the source form of the clauses. But due to the existence of DBOP-predicates,

the source form of clauses is to be retained somehow. So there are conceptually two Prolog databases in the system: the CDB (Code DataBase) and the SDB (Source DataBase), which represent the same programs. They differ only in implementation techniques and in the form of representation. While the SDB can be implemented either virtually or physically, the CDB must be implemented physically.

In our opinion, the execution mechanisms of non-logical components are composed of the following parts:

- Methods to implement and manage the CDB;
- Methods to implement and manage the SDB;
- Methods to implement the DBOP-predicates;
- Methods to implement the normal built-in predicates.

The normal built-in predicates refer to the built-ins other than DBOP-predicates in standard Prolog. Some of them (e.g., =, not, var, atomic, etc.) can be compiled into instructions of the WAM, while the others are to be implemented with special routines. The implementation methods of most of these built-ins are either relative simple or are similar to that in interpreter-based systems (with the exceptions of cut, call, and so on). We will not discuss this part in the paper.

3. SEMANTICS OF MODIFIABLE PROCEDURE CODE

The idea of modifiable code is proposed in order to implement the CDB. As the modifiable code can be changed during runtime, problems arise as to how to define the semantics of modifiable procedure code. This is not a particular problem in compiler-based systems. In fact, many interpreter-based systems have their own ways of implementing assert and retract, which may lead to the situation that executions of the same program containing assert and/or retract on different systems result in different answers! CHRIS MOSS illustrates in [3] that for a program which test assert and retract he gets 9 different results out of 12 systems. This implies that to make transportations of Prolog programs among different implementations easier, efforts should be made to arrive at one or two agreed-upon definitions of the semantics. We try to give a reasonable definition of the semantics in our model.

Clauses are the basic units for the modifications of procedure code. That is to say, a clause can either be inserted into or deleted from a procedure, but the essential code for the clause usually does not change (i.e., the semantics of the clause is not changed). So defining the semantics of modifiable procedures is just to define the Candidate Clause Sequences (which is called CCS in the remaining of the paper) for procedure calls. There are two possible semantics which we think reasonable. In the following discussing, a procedure is regarded as a sequence of ordered clauses.

3.1 The First Choice of Semantics: Fixed CCS

According to this semantics, the CCS for a procedure call is the clause sequence contained in the procedure at the instant of entering the procedure. From the viewpoint of the procedure call, the procedure code will never change even if it has actually been changed through the execution of DBOP-predicates. So every call to the procedure sees a frozen code. The simplest way to implement this semantics is to make a copy of the procedure code for every call. But its time-space overhead is too much.

The classification of semantics here is similar to that described by T.G.Lindholm and R.A.O'Keefe in [4]. The "virtual copy" method proposed by them is an efficient one to implement the fixed CCS semantics (or the Logical View

of dynamic code in terms of their classification).

3.2 The Second Choice of Semantics: Dynamic CCS

With this semantics, the CCS for a procedure can be dynamically changed during its execution. It is not predictable whether a clause is a candidate until it has been chosen to execute. The side effects of DBOP-predicates can immediately be reflected onto the execution of calls to the modified procedure, which might be informally considered to be more real-time. This is the semantics adopted in our model.

For the convenience of further discussing, we introduce the successor function for clauses:

succ(t,C) where t is time, C is clause.

Its definition is as follows:

(1). When C is a non-deleted clause, the value of succ(t,C) is the first non-deleted clause after C in the clause sequence.

(2). When C is a deleted clause, succ(t,C)=C'=succ(t',C), where t' is the time when C is deleted. succ(t,C) usually does not change unless C' is also deleted from the sequence later, in which case we'll have:
succ(t,C)=succ(t,C')=succ(t'',C'),
where t'' is the time when C' is deleted.

(3). When C is the last clause, succ(t,C) is null.

Recall that when executing a procedure call, candidate clauses are tried in turn to match the clause head with the calling goal until a matching clause is found or a failure occurs. In the former case, the clause is selected to execute, producing an instance of the clause which corresponds to the procedure call. When the instance fails, backtracking occurs, and the above steps are repeated. We can see from the process that for a called procedure, there is a one to one correspondence between procedure calls and clause instances.

The following is our definition of the dynamic CCS semantics:

(1). The CCS for a procedure call just commenced is the current sequence of non-deleted clauses contained in the called procedure.

(2). If, at time t, a procedure call (e.g., pcall) corresponds to an instance of clause C, then the CCS (remaining part) for the procedure call is:
C0, C1, ..., Cn
where, C0=succ(t,C)≠nul,
Ci=succ(t,Ci-1)≠nul (i=1,2,...,n),
and succ(t,Cn)=null.

(3). When a clause is deleted, all instances of the clause should finally fail.

According to (2), if the position of the clause to be inserted or deleted by a DBOP-predicate at time t is before succ(t,C), this modification has no effects on the procedure call pcall.

4. A SCHEME TO IMPLEMENT THE CDB

4.1 Extensions to the WAM

4.1.1 A Brief Introduction to the WAM

The WAM can be considered to be composed of three parts: the data areas, the machine state and the instruction set. The data areas include: the code area, the heap, the local stack, the trail and the push down list. The code area contains the codes for Prolog programs, which is a (pseudo) static area. The

heap contains structured data which are constructed during runtime by explicitly copying functors and arguments into consecutive memory cells. The local stack contains choice points (which are created to effect backtrackings) for procedures and environments for clauses. The trail contains references to variables which must be restored to unbound on backtracking.

The machine state is defined by a number of registers, including a set of argument registers and some pointer registers which contain pointers pointing into various data areas, such as the program pointer P, the last environment pointer E, the backtracking point pointer B, and so on.

The instructions of the WAM can be classified into 6 groups: the get instructions, the put instructions, the unify instructions, the indexing instructions, the choice point instructions and the control instructions. In the WAM, argument values in calling goals are passed through the argument registers. The put instructions are responsible for loading the argument registers, while the get instructions are responsible for matching against the values in the argument registers. The unify instructions are responsible for performing unifications with arguments of structures.

The indexing instructions and the choice point instructions are the ones which most closely relate to the management of the CDB. The function of indexing instructions is to selectively try clauses of a procedure (rather than try each one in turn), thus filtering out those clauses whose heads definitely cannot match the calling goal. There are three indexing instructions:

```
switch_on_term VLbl,CLbl,LLbl,SLbl
switch_on_constant N
switch_on_structure N
```

The first one causes a multiway branch to the code block with starting address VLbl, or CLbl, or LLbl, or SLbl, depending on whether the first argument of the calling goal is a variable, or a constant, or a list, or a structure. The latter two are used for second layer multiway branching in the cases that the first argument is a constant or a structure respectively.

Choice point instructions are responsible for managing choice points. Try_me_else is the first instruction of the first clause of a procedure. Its main function is to create a choice point for the procedure. It fills in the NCCP(next candidate clause pointer) field in the choice point with the address of next clause. Retry_me_else is the first instruction of intermediate clauses. It updates the NCCP field in the current choice point to point to next clause. Trust_me_else fail is the first instruction of the last clause. Its function is to remove the current choice point.

During compilation, a clause is compiled independently to a clause code block(or CLB for short) whose instructions are stored in a stretch of consecutive memory cells in the CDB. The CLB's for a procedure are linked together through choice point instructions, constituting the Variable Code Block which is the basic block. Then index is produced for the procedure by the compiler in order to increase the execution efficiency of the code. For more details about the compilation of Prolog, refer to [5,6].

4.1.2 Problems with the WAM

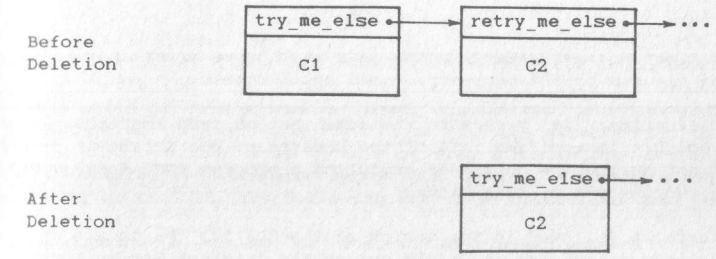
Unfortunately, the original WAM is only an execution model of logical components of Prolog. Thus it should be extended and attached with execution mechanisms of non-logical components.

The WAM cannot implement any one of the two semantics discussed above. The main reasons for this are:

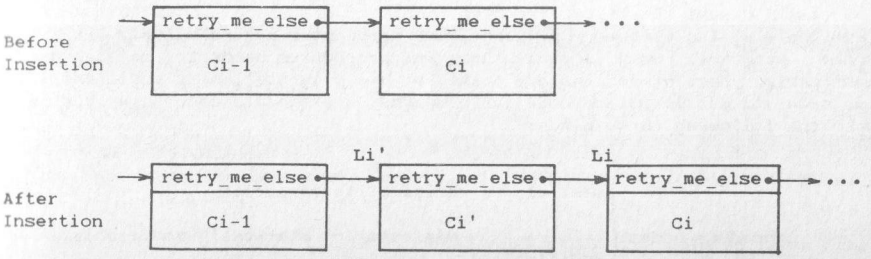
(1). Choice points are removed too early, which may lead to inconsistent semantics. If a clause is asserted as the last one of a procedure, any procedure calls corresponding to instances of clauses other than the original last one may well find the new clause. But the ones corresponding to instances of the original last clause cannot because their choice points have already been removed.

(2). Choice point instructions for the first, the last, and intermediate clauses are different. When the first clause C1 is deleted, the original second clause C2 becomes the first, the choice point instruction of which should be changed to try_me_else (Fig.1 (a)). But for the procedure calls corresponding to the instances of C1, its choice point has already been created by the try_me_else in C1, execution of another try_me_else in C2 after failure of C1 will cause an error.

(3). The execution control links between adjacent clauses are direct references. When a clause Ci' is inserted into an intermediate position of a procedure (Fig.1 (b)), then for the calls corresponding to the instances of Ci-1, they will skip over the newly inserted clause when the instances fail. For Li(the address of Ci) has already been filled in the NCCP of their choice points. This does not satisfy the dynamic CCS semantics. On the other hand, for calls correspoinding to instances of clauses preceding Ci-1, they will certainly find the inserted clause, which contradicts the fixed CCS semantics.



(a)



(b)

FIGURE 1
Insertion and deletion of the Warren code

4.1.3 Extensions to the WAM

To implement the dynamic CCS semantics, we made some extensions to the WAM.

(1). An one-bit flag CTRYF is added to the machine state which is used to determine the function of ctry_me_else instructions. The structure of a choice point is modified to include a corresponding field to save the value of CTRYF. The CTRYF is reset each time a procedure call instruction is executed, and is recovered from the saved value in the current choice point whenever backtracking occurrs.

(2). Two new choice point instructions are added: ctry_me_else LA,L and trust_fail. The L and LA are the address of next clause and the address of L respectively. Ctry_me_else always occurs as the first instruction of every CLB of a dynamic procedure. Its function depends on the value of CTRYF. When CTRYF is 0, it represents that the current ctry_me_else instruction is the first one, in which case the ctry_me_else functions similarly to try_me_else. When CTRYF is 1, it means that the current ctry_me_else is not the first one, in which case the ctry_me_else functions similarly to retry_me_else. Trust_fail is always the last instruction of a procedure, its function is to remove the current choice point and to cause backtracking.

(3). The code area is managed as a heap which can be dynamically updated and garbage collected.

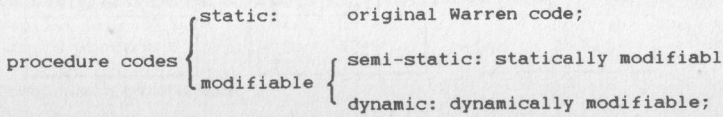
4.2 Classification of Procedure Codes

After analysing various cases of updates of Prolog databases, we find that there are two kinds of modifications of a procedure:

- Dynamic modification: it refers to the case that there exist in the system instances of clauses of the procedure under modification.

- Static modification: it refers to the case that no such instances exist when the procedure is modified. All of the updates of the databases made by the users not during the course of executing a program and many updates made during runtime are of this kind.

One of the criteria we followed in the design of the CDB is: We should do our best to keep the code as compact and efficient as the original Warren code. It is true that the dynamically modifiable code in our model is inferior to the original Warren code in time-space efficiency. This is because the address filled into the NCCP of a choice point by a ctry_me_else is the indirect address of next clause, and the choice point is deleted only after the code of the last clause has been executed, which costs more local stack space. Nevertheless, the frequency of using DBOP-predicates are relatively low in most Prolog programs. Only a small number of procedures needs to be dynamically modifiable. Most procedures are static or are only statically modified. Thus, in order to get optimized code for programs, procedure codes are classified into the following three kinds:



These three kinds of code are similar with each other. They only differ in a few instructions and in the information they contained which support the management of the CDB. At present, the dynamic code in our model must be unindexed. But the semi-static code can take advantages of indexing. An indexing state flag is associated with each procedure indicating its indexing

state. A procedure is compiled into a semi-static procedure by default unless it is declared by the programmer to be static or dynamic.

4.3 Garbage Collection of Code Space

Modifications of procedure code include the modifications of Variable Code Block and the modifications of indexing code blocks. Both the Variable Code Block and the indexing code blocks are managed as linked lists. For details about the basic operations on these blocks, refer to [9].

For semi-static procedures, the memory space of the deleted CLB's can be reclaimed immediately. This is because there exist no instances of the corresponding clauses in the system, hence no references to the deleted CLB's. But for dynamic procedures, the situation is complicated. The space of the deleted CLB's cannot be reclaimed unless all the references to the blocks have been removed.

We have considered two schemes to implement the garbage collection. The first one is described in [8], which is based on the assumption that only one linked list of CLB's is maintained. The list contains all the non-deleted CLB's and the deleted CLB's. This scheme has the disadvantage that there are two kinds of references to a CLB: claiming references and preclaiming references. Overhead is incurred to deal with preclaiming references. Besides, the linked list may become very long after performing lots of assert and retract operations, which will increase the time required to search down the list.

The second scheme is the one currently adopted. It requires two linked list of CLB's be maintained. One contains all of the non-deleted CLB's, the other contains all of the CLB's which have been deleted but whose memory space has not been reclaimed. The information contained in each CLB which supports the implementation of this scheme includes: CLB.claimed, a one-bit flag which, when set, indicates the corresponding clause has been selected for execution and is still pending; CLB.deleted, a one-bit flag which indicates whether the clause has been deleted from the procedure; CLB.length, a two-byte field which gives the length of the CLB. The following are three primitives used in this scheme.

(1). Claiming a clause: a clause is claimed when it is selected for execution for the first time.

```
Algorithm:
  if (CLB.claimed=0)
      then {  Set CLB.claimed;
              Push a pointer to the CLB in the trail; }
```

(2). Unclaiming a clause: a clause is unclaimed when it is no longer needed in the execution, which happens when the system backtracks past it. An entry can be found in the trail at this time which points to the CLB of the clause.

```
Algorithm:
  if (CLB.deleted=1)
      then Deallocate the CLB;
      else Clear CLB.claimed;
```

(3). Terminating a CLB. It is invoked when a CLB is deleted from the procedure.

```
Algorithm:
 if (CLB.claimed=0)
    then Deallocate the CLB;
    else {   Set CLB.deleted;
             Chain the CLB into the linked list of deleted CLB's;  }
```

5. CONSIDERATIONS ON THE IMPLEMENTATION OF THE SDB

The source form of clauses is stored in the SDB, which can be implemented either physically or virtually. Physical SDB refers to the case that a real Prolog source database, like the one in an interpreter-based system, is maintained; while virtual SDB refers to the case that no such a database is maintained, and the source form of clauses is dynamically constructed by means of decompilation or other methods.

Clocksin pointed out in [7] that there are three possible ways to retain the source form of clauses:

(1). Retain a copy of source clauses: The source form of clauses is physically stored in a data area in the system.

(2). Decompilation: the source form is dynamically constructed by decompiling the object code.

(3). Meta clause method: whenever a clause is compiled, a meta clause which includes the clause as one of its argument is compiled. Both the code for the clause and the code for the meta clause are loaded into the CDB. The source form of the clause can be constructed by execution of the meta clause.

We have made an analysis and comparison among them. The characteristics of each method are summerized below:

The first method:

Advantages:

- Each clause is compiled only once, which does not increase the time required to compile a program.
- The CDB and the SDB are relatively independent of each other, which provides much more flexibilty for implementation;
- It supports the optimized implementation of DBOP-predicates and the implementation of execution driven compilation.

Disadvantages:

- The memory space used is about the same as that in the third method;
- A dedicated SDB manager is required;

The second method:

Advantages:

- Each clause is compiled only once.
- Nearly nothing other than code needs to be stored.

Disadvantages:

- The design of the decompiler has something to do with the compiler.
- The optimization of code is constrained more or less;

The third method:

Advantages:

- Clause searching and matching can be done by the usual mechanisms of the WAM.

Disvantages:

- Each clause is compiled twice, which doubles the compile time of a program;
- Besides the code for clauses, the CDB contains the code for meta clauses. So the requirement of storage doubles.

Though all the three methods are feasible, we believe that the first two are better ones in the framework of the WAM. The reasons are following. Like the decompilation method, the meta clause method implements a virtual SDB. But the time required to compile and the memory space required to store a program when

the meta clause method is adopted is about twice as much as that when the decompilation method is used. One the other hand, the Warren code has the unique feature that it can be efficiently decompiled. Kevin A. Buettner has designed a fast decompiler for the Warren code [2], which showed that decompilation is a good method to implement a virtual SDB. Though in the first method the requirement of memory space is about the same as that in the third method, the compile time of a program is not increased. What is more important, the first method is the only method to implement a physical SDB. A physical SDB is more powerful than a virtual one. The execution driven compilation technique(In a system which uses such a technique, a procedure is compiled only when it is called for the first time) and the optimization discussed in the next section cannot be implemented without a physical SDB. So we have adopted the first method in our model. Our algorithms to implement a physical SDB is described in [10].

The second method has the extra advantage that it supports easy implementation of debuggers. Whereas in both the first and the third method, it is relatively difficult to construct the source form of a single subgoal which is often required to be printed out when debugging. We think that the best method to implement a powerful SDB is the combination of the first two methods.

6. AN OPTIMIZATION FOR THE IMPLEMENTATION OF DBOP-PREDICATES

In many cases, the source terms inserted or deleted by assert, retract are only used as data structures (e.g., global variables, partial results), which need only be stored in the SDB. They do not need to be compiled. To optimize the implementation of DBOP-predicates, two operation modes are defined: SINGLE and DOUBLE. SINGLE mode predicates only operate on the SDB, DOUBLE mode predicates operate on both the SDB and the CDB. When a term is to be asserted as a data structure, SINGLE mode assert can be used to do this.

In the standard Prolog, the implied operation modes for DBOP-predicates are as follows:

listing, clause ---- SINGLE
assert, retract, etc. ---- DOUBLE

We were not aware of the operation modes before because in interpreter-based systems, no CDB's exist. In our model, we have made the CDB and the SDB visible to the users. Both SINGLE mode and DOUBLE mode asserts and retracts(and their variants) are supplied to the programmers. However, the default mode for a DBOP-predicate which can have two modes is DOUBLE, which is consistent with the conventioal semantics of the predicate in the standard Prolog.

The annotations used in the model is similar to that in Concurrent Prolog[11] and Parlog[12] in the sense that they are all non-logical, and that they all supply the programmers with facilities to give control information in their programs which is useful for raising execution efficiency. But differences do exist. The annotations in our model are mainly for the purposes of implementing the DBOP-predicates which are themselves non-logical. So if the BDOP-predicates are allowed to exist in an implementation, there will be few reasons for rejecting them.

With the supports from the CDB and the SDB, it is easy to implement the DBOP-predicates. The algorithms for some DBOP-predicates are given in [8].

7. CONCLUSIONS

In the paper, an execution model for the non-logical components of Prolog in WAM-based systems is proposed which is composed of four parts: the code database, the source database, the DBOP-predicates and the normal built-in

predicates. The decomposition of a Prolog database to the CDB and the SDB makes it clearer to get a correct and neat implementation. Semantics of modifiable procedure code and schemes to implement the CDB and the SDB are discussed. The model has the advantage of implementing consistent semantics of modifiable code while keeping relatively high execution efficiency. No overhead for modifiable code would be incurred if one do not modify procedures, in which case the performance of the model will be about the same as that of the WAM.

The study of the execution mechanisms of the non-logical components under the framework of WAM is the first step towards the final goal of our research project which aims at the implementation of a high-performance Prolog machine which will be based on the WAM. In order to test the model proposed in the paper and to provide a good environment for further studies, we have implemented a simulation-based experimental system in C on VAX-11/780. The system is composed of 7 parts: the Prolog compiler, the assembler, the disassembler, the simulated WAM, the CDB, the SDB and the interactive command executive. It has a very friendly user interface. Currently we are developing a compiler-based Prolog system on the basis of the experimental system.

REFERENCES

[1]. Warren, D.H.D., An Abstract Instruction Set, Technical Note 309, SRI International, Oct., 1983.

[2]. Buettner, K.A., Fast Decompilation of Compiled Prolog Clauses, Proceedings of the Third International Conference on Logic Programming, July, 1986.

[3]. Moss, C., CUT&PASTE - defining the impure Primitives of Prolog, Proceedings of the Third International Conference on Logic Programming, July 1986.

[4]. Lindholm, T.G. and O'Keefe, R.A., Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code, Proceedings of the Fourth International Conference on Logic Programming, 1987.

[5]. Roy, P.V., A PROLOG Compiler for the PLM, Master's Thesis, Computer Science Division, University of California, Berkerley, Aug., 1984.

[6]. Bowen, K.A., Buettner, K.A., Cicekli, I. and Turk, A.K., The Design and Implementation of a High-Speed Incremental Portable Prolog Compiler, Proceedings of the Third International Conference on Logic Programming, July, 1986.

[7]. Clocksin, W.F., Implementation Techniques for PROLOG Databases, Software - Practice and Experience, Vol. 15 (7), July, 1985.

[8]. Chenxi, Z., Yungui, T., Liangliang, L. and Yunfa, H., An Approach to the Implementation of Prolog Code Database and Source Databases in Compiler-Based Systems, Proceedings of the Fourth International Symposium on Logic Programming , Aug., 1987.

[9]. Chenxi, Z. and Yungui, T., A Scheme for Managing Prolog Code Databases, Technical Report, Changsha Institute of Technology, TR-87-6016, June, 1987.

[10]. Chenxi, Z. and Bo, L., Schemes for Implementing Prolog Source Databases, in Chinese, Submitted to the Third National Conference on Logic Programming, Aug., 1987.

[11]. Shapiro, E.Y., A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report, TR-003, Feb., 1983.

[12]. Clark, K.L. and Gregory, S., Parlog: Parallel Programming in Logic, Research Report DOC 84/4, Imperial College, London.

IMPLEMENTATION AND PERFORMANCE OF A COMPLEX VISION SYSTEM ON A SYSTOLIC ARRAY MACHINE*

Ed Clune, Jill D. Crisman, Gudrun J. Klinker, and Jon A. Webb

Computer Science Department and The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Complex vision systems are usually quite slow, requiring tens of seconds or minutes of computer time for each image. As the complexity and experimental nature of the system increases, the speed is especially low, since all components of the system must be optimized if the system is to show good performance. The FIDO system, a stereo vision system for controlling a robot vehicle, has existed for a number of years and has been implemented on a number of different computers. These computers have ranged from a DEC KL10 to the current implementation on the Warp machine, a 100 Million Floating-point Operations Per Second (MFLOPS) systolic array machine. FIDO has shown an enormous range in speed; its ancestor took 15 minutes per step, while the Warp implementation takes less than 5 seconds per step. Moreover, while early versions of FIDO moved in slow, start-and-stop steps, FIDO now runs continuously at 100 mm/second. We review the history of the FIDO system, discuss its implementation on different computers, and concentrate on its current Warp implementation.

1. INTRODUCTION

Sophisticated vision algorithms are usually disappointingly slow when they are put together into a complete system. This is especially true of systems created for research purposes, since speed is usually not a primary concern and much effort must be spent on working out ideas in a user-friendly environment, which is not conducive to high speed algorithms. However, even in a research environment, reasonable speed is desirable. One reason for this is that it is difficult to debug an algorithm that takes minutes to compute one meaningful result; not many debugging runs can be made in a day. Another reason is the difficulty of using the algorithm in a non-simulated environment, for example to control an actual robot vehicle; if the algorithm takes minutes to run for each step, it is not practical to debug given time constraints imposed by the physical environment. Finally, it is not possible to integrate an algorithm into a working vision system, which may use multiple sources of inputs, if the speed of one algorithm is much lower than the others.

One sophisticated vision system that has received much attention over the years, and which continues to be developed, is the *FIDO* (*F*ind *I*nstead of *D*estroy *O*bjects) vision system. FIDO is a stereo vision navigation system used for the control of robot vehicles; it includes a stereo vision module, a path planner, and a motion generator. This system is descended from work done by Moravec at Stanford [1]. After Moravec came to Carnegie Mellon in

*The research was supported in part by Defense Advanced Research Projects Agency (DOD), monitored by the Air Force Avionics Laboratory under Contract F33615-84-K-1520, and Naval Electronic Systems Command under Contract N00039-85-C-0134, in part under ARPA Order number 5352, monitored by the US Army Engineer Topographic Laboratories under contract DACA76-85-C-0002, and in part by the Office of Naval Research under Contracts N00014-80-C-0236, NR 048-659, and N00014-85-K-0152, NR SDRJ-007.

1980, work was done by Thorpe and Matthies [2; 3], who gave the system its name. More recently, work has been continued by the authors of the present paper, as well as others. This vision system is unusual in its longevity and in the range of speed over its span of development: Moravec's original algorithm, which was heavily optimized (though different in many important ways from the FIDO algorithm), took 15 minutes to make a single step while running on an unloaded DEC KL10; the Vax 780 implementation ran at 35 seconds per step; the Sun 3 implementation presently at Carnegie Mellon takes 8.5 seconds per step; and the current Warp implementation takes 4.8 seconds per step. This is a factor of 190 in improvement.

We will begin by describing the Warp system, on which the current implementation of FIDO runs, and how it is programmed. Then we will review the history of the system that became FIDO, and how the different parts of FIDO were implemented on Warp. Finally, an evaluation of the Warp/FIDO system is given and a brief description of future implementations.

2. THE WARP SYSTEM

2.1. WARP HARDWARE

A discussion of the Warp hardward is necessary in order to fully understand the Warp implementation of FIDO. This discussion is greatly abbreviated; more detail is available elsewhere [4]. The Warp machine has three components: the Warp processor array, or simply Warp, the interface unit, and the host, as depicted in Figure 2-1.

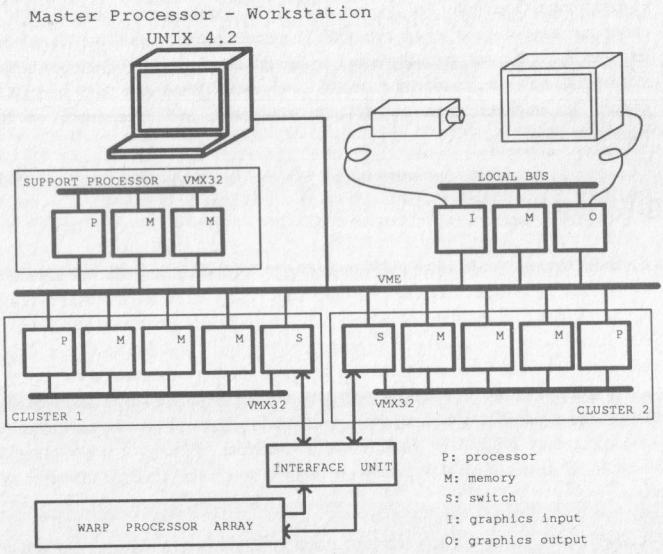


Figure 2-1: Warp host

All of the work on FIDO on Warp so far has been done using the demonstration and prototype Warp machines, which are wire-wrapped machines built according to Carnegie Mellon's design (the demonstration machine was built by Carnegie Mellon; prototypes were built by General Electric and Honeywell). The wirewrapped machine has been superseded by an improved production Warp machine [5], built using printed circuit boards by General Electric. The machine described here is the wire-wrapped machine.

The Warp processor array is a programmable, one-dimensional systolic array, in which all the cells, called Warp cells, are identical. Each cell is a complete computer, with computational units and local data and program memory, except that address generation is normally done on the interface unit, so that addresses, along with data, flow through

the array. Each cell contains two floating-point units: one multiplier and one ALU, each of which can deliver up to 5 MFLOPS. The peak processing rate is 10 MFLOPS per cell. A 4 K-word memory is provided for resident and temporary data storage.

As address patterns are typically data-independent and common to all the cells in low level vision algorithms, full address generation capability is factored out of the cell architecture and provided in the interface unit. Addresses are generated by the interface unit and propagated from cell to cell, together with the control signals. In addition to generating addresses, the interface unit passes data and results between the host and the Warp array, possibly performing some data conversions in the process.

The host consists of several processors: The "master" controlling the Warp array is a Sun workstation which can run Unix to provide a convenient programming environment and provide compatibility with other Carnegie Mellon vision research programs. Data transfer to and from the Warp array, and control of vision peripherals such as cameras and frame buffers, is provided by an "external host" (physically external to the Sun) which communicates with the Sun through a VME bus repeater. The external host consists of three "standalone processors": two of them, called "cluster processors," are used for sending and receiving data during a computation (they can exchange roles as needed) and the third one, called "support processor," is used for controlling peripherals and executing a runtime library. The runtime library allows event sequencing and processing of interrupts from Warp.

2.2. WARP SOFTWARE

The development of programming tools that manage the Warp system efficiently requires the integration and extension of state-of-the-art techniques and further advances in compiler technology.

A basic runtime library has been written for the host as a first attempt at handling the system level issues [6]. The library has mechanisms for allocating memory on the external host and sequencing events on the Master, standalone processors, and Warp array as well as mechanisms for running the Warp array and using peripheral devices.

A compiler exists for a language called "W2," a Pascal-like language in which all the underlying parallelism in the host and within each cell is hidden [7; 8]. Only the parallelism between the cells is seen by the user and must be managed explicitly. An asynchronous send/receive protocol is supported for communication between the cells.

3. FIDO

3.1. HISTORY OF FIDO

The predecessor to FIDO was the Stanford Cart developed by Moravec at Stanford University starting in 1973. The Cart was a vision-guided vehicle that could navigate through a world containing obstacles. The vehicle moved in a series of one meter steps. At the end of each step, pictures were taken to determine the Cart's position by tracking known obstacles in the scene.

The Stanford Cart algorithm and the FIDO algorithm contain the same steps. First obstacles are identified in one image, using a feature detector. Then obstacles in one image are matched with obstacles in the other images. The three-dimensional positions of the obstacles can be now be calculated by using a triangulation technique. Next the motion of the vehicle is planned and executed. After obstacles are found in the new set of images, they are compared to the previous set of obstacles to determine the relative vehicle motion.

The performance of the Stanford Cart algorithm was limited largely by unreliable hardware. For example, each image was digitized 24 times in order to obtain at least one good image, and binocular vision was abandoned in favor of a stereo algorithm that used nine images for greater robustness. The motion of the cart was not accurate, so much time had to be spent determining the cart's exact position after a move. Even with careful optimization, the program needed 15 minutes per step on an unloaded DEC KL10. Despite its slow speed, the Stanford Cart had several successful indoor runs and a successful outdoor run in 1980.

Thorpe and Matthies developed FIDO, starting with simplifications to the Stanford Cart algorithm due to better hardware. Thorpe and Matthies also experimented with the various modules to give a scientific basis to some heuristically chosen parameters. In addition, some modules (for example, the path planner) were re-implemented using faster algorithms. With all of these changes, FIDO was able to run in the laboratory, on the Neptune robot on a Vax/780, at a speed of 30 seconds per half meter step, a speedup of 30 over the Stanford Cart. More than 80% of this time was spent running basic vision routines such as image reduction, correlation, and the interest operator.

Crisman began work on FIDO in May 1985. As part of the Carnegie Mellon Strategic Computing Vision Project [9] FIDO was moved to a Sun Workstation (currently a Sun 3). In addition, Crisman began a general cleaning of the FIDO code, which had become quite cluttered with unused and experimental code. Several improvements were implemented to increase the speed of FIDO, including optimizing the display routines and using arcs in the motion generator. The code was also adjusted to run on the new outdoor robot, Terragator. In an attempt to take advantage of much more accurate motion expected from Terragator, the module in FIDO that determined position by obstacle tracking was removed. This also increased the speed of the algorithm.

3.2. FIDO ALGORITHM

FIDO performs the following steps. First, it takes two pictures of its environment. It then tries to locate all of the obstacles that it knows about in the new pictures. Then it checks to see if there are any other obstacles present that it had not been able to see before. It plans a path to its goal around all of the old and new obstacles. Then it moves one meter along the planned path and stops to re-investigate its environment. This repeats until the goal position is reached.

To clarify the algorithm, the following terminology will be used. An *obstacle* is an identified location in the three-dimensional world where the vehicle is unable to pass. A *feature* is the two-dimensional appearance of the obstacle in the image. FIDO considers only point features. This assumes that physical objects will have enough features in the image to be correctly bounded in three-dimensional space.

Below is a more detailed description of the FIDO algorithm.

- *Digitize*: Two 512×512 images, called the left and right images, are taken from identical cameras with known relative positions. The right image from the previous image pair is saved for the reacquire step.
- *Reduce*: Reduce the two 512×512 images to create an image pyramid as shown in Figure 5-1.
- *Reacquire*: Find the currently known features from the old right image pyramid in the new right image pyramid, using pyramid correlation as explained in Section 5.3.
- *Match Features*: Match features from the previous step between the new right and new left image pyramid, using pyramid correlation. The three-dimensional position of the features are then calculated.
- *Catalog*: The catalog is a list of obstacles known to FIDO. This forms a simple map describing the environment surrounding the vehicle. The new objects that are discovered by the reacquire and matching steps are added to the map. Previously found obstacles that should be seen but are not are discarded, to eliminate spurious obstacles.
- *Plan a path*: Plan a path from the vehicle's current position to its destination that avoids all of the obstacles.
- *Move*: Move one step toward the destination, along the path.
- *Pick new features*: Use the interest operator to find new "interesting" points in the image. These features are added to the features matched in the *Match Features* step.
- *Match new features*: Find the new features from the new right image, in the new left image, using pyramid correlation.

4. IMPLEMENTATION OF FIDO ON WARP

In the summer of 1984, Dew, Chang, Matthies and Thorpe designed a new version of the FIDO system to run on Warp, which was then in its initial design phase. They identified the most time consuming parts of FIDO to be the major vision algorithms (correlation, interest operator, and reduction), which were considered to be suitable for

implementation on a systolic array such as Warp. They redesigned FIDO to run on a Warp multiprocessor system, based on an early design that used the Aptec bus [10] as the external host [11]. The external host was later changed to be the one described in section 2.1, largely for reasons of programmability.

Once the initial design was done, implementation of FIDO on Warp proceeded in three steps:

- *Step 1* – Implement the three vision modules of FIDO for the Warp array, providing a simple demonstration program to show that the implementation works. This was done by Klinker using Warp microcode on a demonstration Warp system.
- *Step 2* – Reimplement the vision modules on the prototype Warp system. This was done by Clune using W2. At the same time, exploit the external host to overlap computation on the Warp array with preparation and post-processing of the data in the cluster processors, achieving more parallelism.

The steps above use the most powerful part of Warp – the Warp processor array – to speed up the FIDO system. However, it is also possible to use Warp's external host to exploit algorithm-level parallelism and get further speedup. This approach can sometimes give a large speedup in the system, since in some cases significant portions of the code can be run in parallel with the rest of the system, effectively executing that code for free. Therefore, we formulated *Step 3*: Make efficient use of the multiprocessor host system. To some extent, this step has already been accomplished, but there is still room for exploiting algorithm-level parallelism in FIDO, as we shall discuss later.

Section 5 describes the initial implementation of the vision modules on Warp using microcode (Step 1). Section 6 describes the current implementation of FIDO on Warp in W2 (Steps 2 and 3).

5. FIDO ON THE DEMONSTRATION SYSTEM

FIDO was first implemented on the Warp demonstration system, which included a Sun 2, an interface unit, and two Warp cells. In addition to being the first work on FIDO on Warp, this step showed in the demonstration system that Warp can be used for vision applications. Moreover, it allowed us to test algorithm decomposition techniques and to test the Warp hardware with realistic algorithms. Once the initial implementation was done, it was ported to the ten-cell prototype system and modified to use the cluster processors for input and output. This implementation was then superseded by new software, as discussed in the next section. Here we describe the vision modules in FIDO in detail and give their timings on the demonstration system and first prototype implementation.

5.1. IMAGE PYRAMID GENERATION

The image pyramid used in FIDO consists of 7 levels, starting with a 512×512 image at level $x1$ and ending with an 8×8 image at level $x7$, as shown in Figure 5-1. Areas of 2×2 pixels are replaced by one pixel in the next higher level of the pyramid. The new pixel value is computed by averaging over a window in the higher resolution image to produce a one pixel result in the lower resolution image. The simplest averaging is to take a 2×2 pixel area and average it to one pixel. The initial implementation on the Warp array used overlapping 4×4 windows, which gave slightly better results than 2×2 windows.

5.1.1. MAPPING THE ALGORITHM ONTO WARP

The pyramid generation algorithm has been implemented in Warp microcode in a systolic scheme, as suggested by Kung for convolution-type algorithms [12].

The pyramid generation algorithm was planned to fit the ten cell Warp array. The algorithm required the processors to accumulate and normalize 16 pixels in a 4×4 window to produce one reduced pixel value. This was mapped onto the Warp array as nine modules, with the first eight each adding two new pixel values to the accumulated partial sum, and the ninth module normalizing the result. The second, fourth and sixth module also stored the partial results until the necessary pixels from the next row underlying the 4×4 window had arrived at the module. The new data and the partial results were then sent together to the next module.

Because the demonstration Warp system consisted of only two cells when the micro-programmed version of the pyramid generation algorithm was implemented and tested, we had to face the problem of mapping nine conceptual,

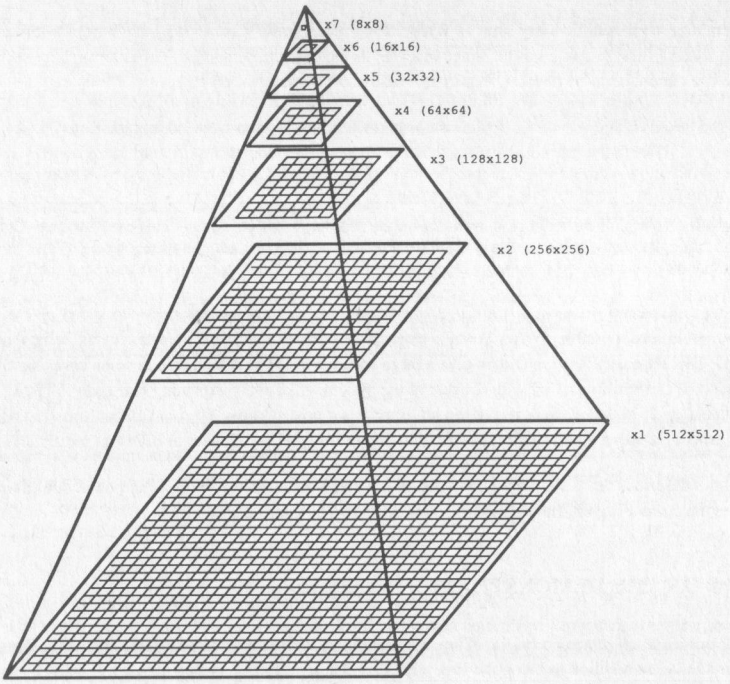


Figure 5-1: Image Pyramid

algorithmic modules onto two physical cells. This was achieved by having the first cell run the first four modules and the second cell run the remaining five modules. Each cell switched between consecutive modules whenever eight new partial results were produced. Thus, the pixels were processed in batches of sixteen at a time. Two extra pixels had to be sent with each batch, due to the overlapping kernel.

5.1.2. MICROCODE TIMING OF THE PYRAMID GENERATION ALGORITHM

Since the entire processing in each module consisted of adding two numbers and passing them on, each module needed two processing cycles to perform its task on a given pair of input pixels. The modules could thus start to operate on a new window position every two cycles. Accordingly, the modules could perform their tasks as fast as the data could be sent to the cells. On an image with $m \times n$ pixels, the nine modules thus needed essentially $m \times n$ cycles to reduce an $m \times n$ image into an $\frac{m}{2} \times \frac{n}{2}$ image, plus a few initial cycles for startup. To generate the described image pyramid, consisting of seven levels, this evolved to $\sum_{l=1}^{6} \frac{512}{2^{l-1}} \times \frac{512}{2^{l-1}}$ cycles, which is

```
341401  cycles   ~ 0.07  seconds      on 9 cells
1365358 cycles   ~ 0.27  seconds      on 2 cells
```

A simpler sequential algorithm (with non-overlapping reduction windows) took about one second on a Vax/780. Nine Warp cells thus provided a speed-up of 14, which is relatively small. The Warp implementation of the pyramid generation algorithm was communication intensive: it used the adder effectively only half of the time (in every other row). It did not use the multipliers at all (except for a normalization). Each Warp cell was used as a 2.5 MFLOPS machine, for 25 MFLOPS from the array. This explains the relatively small speed-up of the pyramid generation algorithm. Note that adding more cells would not increase the speed since this would not reduce the communication requirement.

5.2. INTEREST OPERATOR

"Interesting points" are those points which can be localized well in different pictures (for example corners). The image intensities change rapidly in all directions for a point that is "interesting."

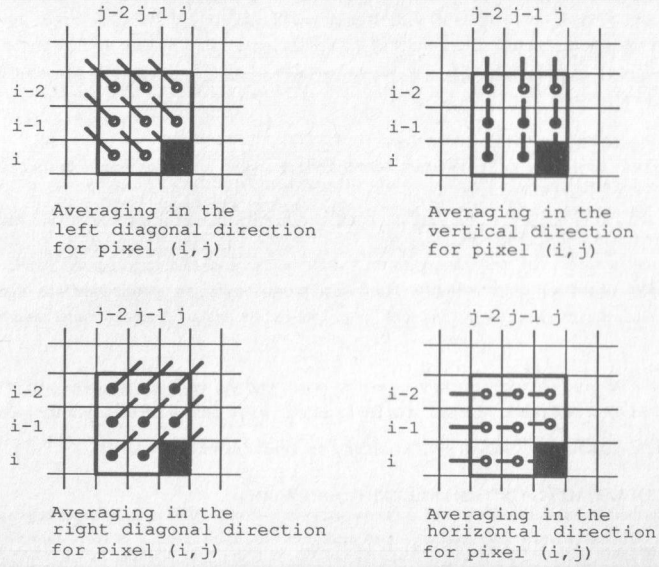


Figure 5-2: Calculation of the interest operator

Interesting points are found using an interest operator, which is largely unchanged from Moravec's Stanford Cart work. The interest operator takes squared pixel differences around a pixel in the vertical, horizontal and both diagonal directions and accumulates them (separately for each direction) in the 3×3 neighborhood of the current pixel [2; 11], as shown in Figure 5-2. For the current pixel to be an interesting point, all four accumulated differences must be large. Their minimum gives the interest value of the current pixel. The interest values are locally maximized in one hundred subimages that are arranged in a 10×10 grid. The maxima of all subimages are stored in a list of decreasing interest values. This gives a set of points, distributed across the image, which can be localized in the image for matching.

Only the first part of this algorithm, computing the accumulated squared pixel differences in all four directions for every pixel, was implemented in microcode on the Warp array. In the demonstration system, processing stopped here. Later, we implemented the minimization, maximization and list processing on the cluster output processor.

5.2.1. MAPPING THE ALGORITHM ON WARP

The interest operator does not have a good partitioning into modules with similar timings. We thus did not try to implement it in a systolic scheme, as was described for the pyramid generation algorithm in the previous section. Instead, we used the *input partitioning* model [13] in which not the algorithm, but the data is divided into equally sized parts. In this scheme, each cell performs the complete algorithm on a portion of the data. An $m\times n$ image is divided into c vertical stripes to be processed on c different cells. For the interest operator, the stripes had to overlap by 4 pixels, due to the width of the operator window. Thus, every cell ran on $m\cdot(\lceil\frac{n}{c}\rceil+4)$ pixels. The systolic communication facilities were then used like a "bus": each cell received data from the previous cell and sent it to the next cell. The host sent the data interleaved such that each cell could use every c^{th} pixel for itself. At the beginning of every new iteration, c new pixels were sent over the "bus." The offset between programs that ran on

neighboring cells was two cycles such that each cell started a new iteration exactly when a new pixel arrived.

Note that, since the programming scheme of the interest operator was organized around partitioning the data and not the algorithm, this algorithm could be easily adapted to run on any number cells. In the demonstration system, the algorithm ran on two Warp cells, computing the interest value for every pixel of a 256×256-sized image (level *x*2 of the right image pyramid) in two parallel vertical stripes each consisting of 256×132 pixels. It was a matter of changing a few constants that indicated the width of the vertical stripes to provide a version that ran on ten cells, when the ten cell Warp became available. The ten cell version divided the image into ten vertical stripes, each consisting of 256×30 pixels.

The algorithm iterated on a sequence of two steps:

1. Get the next pixel, compute the difference between the new pixel and its neighbors in four directions, and square the differences.
2. Add the squared differences to the partially computed interest values of the nine pixels whose 3×3 windows overlap at the current pixel.

Within each of the two steps, the code was optimized using software pipelining to use all facilities, such as the multiplier and adder, of the cell simultaneously. For ease of programming, no optimization was organized between the steps. This made the innermost loop 65 cycles long, whereas the ideal algorithm would have needed only 40 cycles.

The algorithm must store the most recent two rows of pixels and the most recent three rows of partial interest values. Thus, $(2+3\cdot4)\cdot\lceil\frac{n}{c}\rceil+4$ pixels were stored in the local memory of each Warp cell. For the given memory size of 4096 K words, a maximal row width of $n=256$ columns per cell could be allowed.

5.2.2. MICROCODE TIMING OF THE INTEREST OPERATOR

The interest operator ran in $(m-1)\cdot((\lceil\frac{n}{c}\rceil+4)\cdot65+21)+30 \;+\; c\cdot(\lceil\frac{n}{c}\rceil+4)$ cycles. For a 256×256 image, the algorithm takes

```
 502935  cycles   ~ 0.10  seconds      on 10 cells
2193549  cycles   ~ 0.44  seconds      on  2 cells
```

The sequential algorithm ran in about 2.65 seconds on a Vax/780. Ten Warp cells provided a speed-up factor of 26.5. The adder was the most used resource of the interest operator. It was used in 40 out of 65 cycles of the innermost loop. The multiplier was barely used (4 multiplications in 65 cycles). The algorithm thus used each cell as a 3.4 MFLOP machine. The addition of more cells would greatly improve the speed. In the described implementation, each cell needed a new pixel every 65 cycles. Thus, maximally 65 cells could have been used in parallel before the interest operator had become I/O limited. It would then have needed $O(m\times n)$ cycles to compute the results.

5.3. IMAGE PYRAMID CORRELATION

For a given pair of images and a given list of interesting points in one image, the correlation algorithm is used to find the corresponding points in the other image. The search for the most likely corresponding points of the interesting points is performed on the image pyramids, starting at the highest level (*x*7:8×8 image). At each level, a 4×4 template around the current interesting point is taken and correlated with an 8×8 search area in the other image pyramid at the same level. The best matching position of the template in the search area determines the position of the search area in the next lower, more detailed, pyramid level [1], as shown in Figure 5-3.

Pseudo-normalized correlation is used, as given by this formula [11]:

$$CORR_{lm}=\frac{S_t-t_{mean}S_1}{t_{var}+(S_2-S_1^2)/16}$$

$$\text{with } S_1=\sum_{i,j=0}^{3} I_{i+l,j+m},\quad S_2=\sum_{i,j=0}^{3}(I_{i+l,j+m})^2,\quad S_t=\sum_{i,j=0}^{3} t_{ij}I_{i+l,j+m}$$

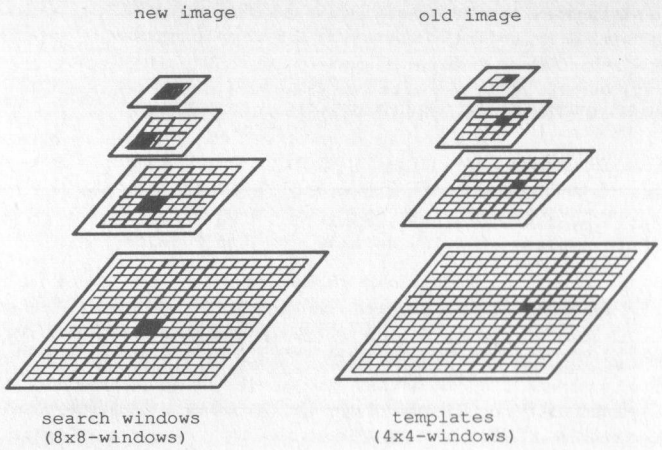


Figure 5-3: Correlation Over an Image Pyramid

where t_{ij} denotes the template element at position (i,j), and $I_{i+l,j+m}$ denotes pixel at position $(i+l,j+m)$ in the image.

In the micro-programmed version of the algorithm, Warp was used to find the optimal correlations of all features for one given pyramid level at a time. First, all templates of a pyramid level were sent. The cells stored the templates and computed their means and variances. Then the search areas of that level were given to the Warp array in the same sequence as the templates. The cells correlated the current template with the current search area and sent the correlation results for every template position to the output cluster. The cluster processor then found the optimal position of each template within its search window and determined the search areas for the next lower level. The process was then repeated for the next lower pyramid level [11].

5.3.1. MAPPING THE ALGORITHM ONTO WARP

The correlation algorithm was implemented in a systolic programming scheme, as for the pyramid generation algorithm. It was designed as nine modules. Each of the first eight modules covered two template elements. The algorithm was designed so that initially, each module received the template elements and stored the respective template elements of each template. The mean and the variance of all templates were computed and stored in the ninth module. Then, in the correlation phase, each module got the pixels of the search areas and the partial sums S_1, S_2, and S_t from its left neighbor and updated the partial sums before it sent them to its right neighbor with the next pair of pixels. As in the case of data pyramid generation, the second, fourth and sixth module stored the derived partial results until the pixels of the next row, underlying the current window position, arrived. The ninth module combined the partial sums and the mean and variance of the current template into a correlation value that denoted how well the template fitted the data in the search area at the current position.

In the demonstration system, the correlation algorithm ran on two cells. The first four modules ran on the first cell, the other five modules ran on the second cell. For every pair of pixels, six additions and four multiplications had to be performed in the first eight modules. In the ninth module, one addition, two subtractions, three multiplications, and two divisions had to be executed. Thus, the adder was the most used resource for the first eight modules (used six times per module run), whereas the multiplier was used most in the last module (five times, so that it was not the bottleneck). It was possible to achieve the optimal speed for the first eight modules, i.e: start a new module run every six cycles, keeping the adder busy all the time. When the modules were run on two cells, however, the ninth module had to share the resources of the second cell with the fifth through eighth module. Since the microcode of the ninth module was very different from the microcode of the other modules (heterogeneous

modules), it was impossible to schedule the facilities of the second cell such that one resource was used in every cycle: the ninth module required that the innermost correlation loop be augmented by twelve cycles (i.e.: two extra module runs).

5.3.2. MICROCODE TIMING OF THE CORRELATION ALGORITHM

The nine modules needed $r \cdot [(6m^2+18) \cdot n+113]$ cycles for n templates and search areas, with each search area being of size $m \times m$, on r resolutions. In FIDO's case, $n=50$, $m=8$, and $r=7$. The algorithm thus ran in

```
141491  cycles   ~ 0.03  seconds      on 9 cells
848946  cycles   ~ 0.16  seconds      on 2 cells
```

The sequential algorithm took about 2.3 seconds on a Vax/780. Nine cells thus provided a speed-up factor of 78. This was a much higher speed-up than that achieved by the pyramid generation algorithm and the interest operator because the multiplier was used in every cycle and the adder was used in every other cycle. Each cell thus ran here as 7.5 MFLOP a machine. The communication facilities were also used in every other cycle. Therefore, the correlation algorithm was a fairly well balanced algorithm. The maximum speed-up would have been reached if 18 cells had been used (due to communication requirements).

5.4. PERFORMANCE OF THE VISION MODULES

Table 5-1: Demonstration Timings

Function	System			Speedup			
	Vax/780	Sun 2	Clusters	Warp: Optimal	Warp: Actual	Maximum (Warp / Sun 2)	Actual
1 pyramid generation	0.5 sec	3.2	0.7	0.3	18.0	11	0.2
2 pyramid generations	1.0 sec	6.4	0.8	0.6	36.0	11	0.2
Interest Op	2.7 sec	13.0	–	0.4	0.9	33	14.0
Correlation	2.3 sec	14.0	–	0.2	3.0	70	4.7

Times for the three vision modules are shown in Table 5-1. The speedup (optimal and actual) of a Sun 2 with Warp over a Sun 2 without Warp is also shown for each module. As mentioned above, only the two cell system was available at this time. In addition, the system software was not fully tuned so that it took approximately 0.3 seconds to call each Warp module.

Most interesting is the speedup of the pyramid generation module on Warp. It actually takes longer to run on the Warp than on the Sun alone. This is because the data flow between the clusters and Warp is unbalanced. Very time consuming manipulations were required to order the data correctly for Warp in this implementation, but the actual pyramid generation on the Warp array is not computationally intensive. The array is virtually starved for data. This is a case where the ordering of data is too complex for Warp (specifically the clusters). A more efficient implementation is for the cluster processors to send the pixels in the order it is stored in memory so that data can flow rapidly into the array, and have the Warp array handle the data reordering itself.

The interest operator and correlation functions did not perform at the optimal speeds either, although they are faster than the comparable Sun functions. If the startup times for the Warp implementations is subtracted (the overhead for startup is much less in the prototype and production Warp machines), the actual times are close to optimal times as shown in Table 5-1.

6. FIDO ON THE PROTOTYPE WARP SYSTEM

The hardware completely changed from the demonstration system to the prototype system described in Section 2. The number of cells in the Warp array increased from two to ten. Just as important for system performance, the master was upgraded from a Sun 2 to a Sun 3 with the MC68881 floating point hardware, and the external host became available. These changes markedly improved the performance of the system.

In addition, the software environment changed radically. Previously, all of the vision modules for the demonstration system were coded directly in microcode, a tedious task. With the prototype system, the W2 compiler became available, greatly simplifying the programmer's task.

We completely reimplemented the vision modules as a result of these changes:

- *Pyramid generation:* This module was reimplemented as a C program to be run on the clusters, since very little computation is done here. This made it possible to do the two pyramid generations in parallel using the two cluster processors. In this implementation non-overlapping 2×2 windows were used instead of the overlapping 4×4 windows in the Warp implementation, to simplify the computation.
- *Interest operator:* This module was reimplemented in W2, without significant change in the algorithm.
- *Correlation:* This module was originally written as a systolic program, but could not be reimplemented in this way because the prototype W2 compiler allowed only homogeneous code. Instead, it was implemented using input partitioning, as in the interest operator.

In addition, preprocessing and postprocessing of the data in modules that send data to the Warp array was implemented as C programs to be run on the clusters, which send and receive data directly to the Warp array. These C programs replaced the standard compiler-generated modules, which transferred data to the Warp array from memory. Use of the clusters in this way exploited some of the parallelism available in the Warp system.

6.1. ANALYSIS OF FIDO PERFORMANCE ON THE PROTOTYPE

The reimplementation of FIDO led to a total system time for one step of 4.8 seconds, which is a large speedup over the original time, but still relatively small compared to the time on a Sun 3 alone (8.5 seconds). Here we analyze how we got this speedup and identify the parts for the system that allow a further increase in performance.

Table 6-1: FIDO Module Times

Module / Function	System: Vax/780 (sec)	Sun 3 (sec)	Sun 3 w/Warp (sec)
Reduce Images	4.8	0.9	0.8
Reacquire Features	13	1.6	0.6
Match Features	7.8	1.6	0.6
Catalog Features	0.1	0.1	0.1
Plan a Path	3.1	0.7	0.7
Pick New Features	2.1	1.6	0.5
Match New Features	7.8	1.6	0.6
3-D Calculation	N/A	0.4	0.9
TOTAL TIME	39	8.5	4.8

6.1.1. PICK NEW FEATURES

'Pick New Features' includes the interest operator function. The actual time for this function to execute is about 0.1 seconds, the same as estimated in Section 4 for the ten cell implementation. Some of the rest of time is spent starting the Warp array (about 25 milliseconds). However, most of the time is spent in post processing. After the

interest operator has been run, some sorting and selecting is done from the resulting data. This is done on one of the cluster processors, which is about 28% slower than the Sun 3 processor, because of a slower clock rate. The effect of this post processing is shown in Table 6-2.

Table 6-2: 'Pick New Features' Times

Function	System	
	Sun 3 (sec)	Warp
Interest Op.	1.3	0.1
Sorting	0.3	0.4
TOTAL TIME	1.6	0.5

The interest operator is sped up by a factor of 10, from one second to 0.1 seconds. This leaves only the sorting and selecting, which was about one-quarter of the time of the original function, but which is 80% of the time in the Warp implementation. The total speedup is approximately three.

6.1.2. CORRELATION MODULES

All of the modules that use the correlation function (e.g. 'Match Features') have less than a factor of three speedup, compared to a Sun 3 alone. A breakdown of the times for Warp is shown in Table 6-3.

Table 6-3: Times for Correlation Modules

Function	Time (sec)
Setup	0.3
Startup	0.2
Correlation	0.1
TOTAL TIME	0.6

As with the interest operator, the implementation of the correlation function on the Warp array reduces its execution time, formerly a large component of the total, to a small factor, and the small amount of time needed to process data for the Warp array on the cluster processors dominates the total. This time included the following:

- Warp startup overhead of 25 ms. In one step, correlation is called seven times, for a total overhead of approximately 0.2 seconds.
- Rearranging data for Warp. Complex addressing is needed to send the image patches from the different pyramid levels to Warp.
- Because of restrictions in the prototype W2 compiler, a fixed number of features must be processed in every correlation, which is set to fifty in this case, although the average number of features in a correlation is approximately half that.

7. ALGORITHM-LEVEL PARALLELISM IN FIDO

It is possible to exploit algorithm-level parallelism in FIDO, as shown in Figure 7-1. Pyramid generation, which takes a large amount of time since it must access every pixel of the original image, can be executed in parallel with calculation of the three-dimensional coordinates of the image features. Similarly, finding new features can be done in parallel with cataloging and path planning and execution. FIDO does not yet take advantage of these sources of parallelism, but our analysis of FIDO so far allows us to make predictions of performance once these approaches have been exploited.

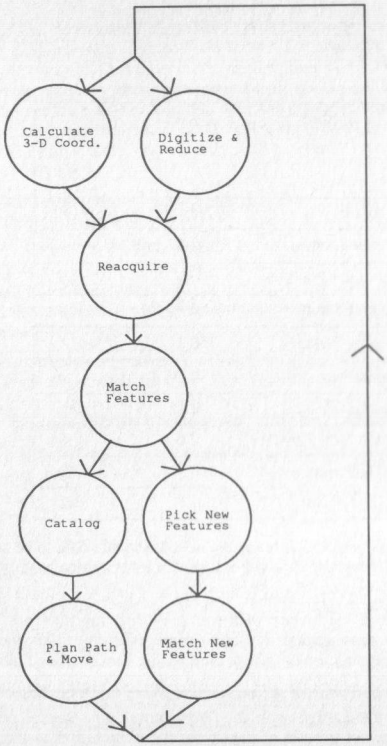


Figure 7-1: Planned FIDO loop with parallelism

These predictions are shown in Table 7-1, compared with the times on the current FIDO system. We see that pyramid generation and cataloging and path planning and execution are done for free. This reduces the total time for FIDO from 4.8 seconds to 3.2 seconds, which is a speedup of 2.6 over the Sun 3 version.

8. USING THE PRODUCTION MACHINE

General Electric has now manufactured a production version of the Warp machine, as mentioned earlier. This machine design was influenced by our experiences with the prototype system, including our experiences with FIDO. We have made a number of modifications to this system, which will improve the performance on FIDO:

- *System overheads reduced.* Overhead for startup of a Warp program is now 5 milliseconds, down from 25 milliseconds on the prototype. This will substantially reduce the overhead of calling a Warp program with a fast execution time. For example, this should reduce the startup overhead in correlation from approximately 0.2 seconds to 35 ms.
- *Heterogeneous code.* In the course of implementing FIDO, we originally implemented the pyramid generation and correlations functions systolically, then reimplemented them using input partitioning when we reprogrammed them using W2. This was due to a restriction in the prototype machine that made it impossible for W2 to support heterogeneous code in a general way. This restriction has now been removed, and we are free to reimplement these modules systolically.
- *Variable loop bounds.* The prototype W2 compiler required that all loop bounds be known at execution time. In FIDO, this meant that the correlation function always processed fifty features, although the average useful number was twenty-five. This implied a doubling of the execution time on Warp, and,

Table 7-1: FIDO Module Times

Module Function	System Sun 3 w/Warp (sec)	Sun 3 w/Warp parallel (est) (sec)
Reduce Images	0.8	–
Reacquire Features	0.6	0.6
Match Features	0.6	0.6
Catalog Features	0.1	–
Plan a Path	0.7	–
Pick New Features	0.5	0.5
Match New Features	0.6	0.6
3-D Calculation	0.9	0.9
TOTAL TIME	4.8	3.2

even more significantly, a doubling of the time for the host to prepare data to be sent to Warp. Allowing variable loop bounds in the W2 compiler should therfore eliminate approximately 0.2 seconds.

- *More flexible programming model.* The production Warp machine supports programs with variable execution time, and makes use of the reverse path in the Warp array (therfore, global communication among the cells that does not pass through the host) more efficient. This means that FIDO can be reorganized to do even more computation on Warp, eliminating host overhead altogether.

- *Hardware improvements.* The hardware of the production Warp machine includes these enhancements:
  - Direct framebuffer communication with the Warp array. A special purpose board has been constructed that allows direct transfer of data from the framebuffer to the Warp array, bypassing the cluster processors. This will allow reimplementation of the pyramid generation step on the Warp array, with each pyramid generation taking approximately 60 ms.
  - Faster cluster processors. The clusters are being upgraded to use microprocessors with a faster clock.
  - DMA from the host. The new processors also support DMA from the host to the Warp array, eliminating a significant bottleneck in feeding data from the host to the Warp array.

9. SUMMARY

We have discussed the history of the FIDO algorithm, and its gradual increase in speed over a period of some 13 years – starting from Moravec's work on its predecessor at Stanford, through the current implementation on Warp. Over this period of time, three things have influenced its speed:

- *The reliability of the sensing and motor control devices available.* With better digitizers, the matching was more reliable, which helped Thorpe reduce the number of images needed for reliable matching. With better motor control, Crisman could completely eliminate the motion parallax step from FIDO. This accounts for about one order of magnitude increase in speed, but much more than this in terms of programming simplicity and ease of use.

- *The speed of the computer hardware.* This effect has been masked by the willingness of different researchers to carefully optimize code for greatest speed: Moravec unwound loops into arrays for maximum speed, for example, and Klinker programmed by hand a machine for which compiling was considered a significant research project. Also, the execution times for early versions of FIDO were never obtained routinely, but only in demonstration runs when everyone else could be removed from the system, which was a significant computing resource for many people. Execution times for more recent versions of FIDO are from more or less routine runs, when only a few if any people were inconvenienced by the desire to get the best time possible. Moreover, the effect of different improvements in hardware varies: the most sudden speedup in FIDO was due not to the introduction of Warp, which required recoding programs, but due to the replacement of a Sun 2 by a Sun 3 workstation.

- *The resourcefulness of the researcher.* A researcher can get more use out of a computer system by placing constraints on calculations to reduce processing. For example, Thorpe and Matthies were able to increase the reliability of FIDO, while reducing the number of images needed, by adding more constraints in the stereo matching. This has accounted for about one order of magnitude improvement in speed. Note that the usefulness of constraints depends on the reliability of the underlying hardware (e.g. sensors) and that any improvement in computational speed of a program can be used to perform more experiments or incorporate other functions into the program.

Warp's potential in the implementation of FIDO is due to several factors, which reflect not only on the design of Warp but also on other special-purpose machines:

1. The Warp array works well for the majority of the computation in FIDO, namely low-level vision computations. Working either in microcode or W2, we never had problems with the Warp array not having enough effective computation power. However, while low-level vision computations form the majority of the computation of FIDO, simply speeding them up is not enough for good speedup of the FIDO system as a whole.

2. The external host is the weakest part of the Warp system. This was known when the host was designed; it was determined once we decided to use industry-standard processors and buses, instead of building our own. In our early versions of FIDO on Warp, this kept us from realizing full use of the Warp array, because of the constraints in rearranging data on the external host.

3. The programmability of the Warp array allowed us to modify our algorithms and programming models to accomodate a regular data pattern from the host. This is important even in the latest versions of the host, which have faster processors and higher data rate, but which can use DMA, which requires a regular address pattern. This flexibility is the main reason we have been able to observe the predicted performance of algorithms in actual Warp runs.

4. W2 makes it possible to experiment with different algorithms, in the context of a research system such as FIDO, while getting good use of the powerful Warp array. As we program more and more of FIDO on the production Warp machine, programmability is essential, especially as it allows us to make use of more complex programming models that use the powerful Warp array more and require less intervention by the relatively weak host.

5. Although the computing power of the external host is small compared to the Warp array, its programmability, and its close integration with the master and the Warp array, makes it an important part of the Warp system. Irregular operations can be mapped onto it as part of pre- and post-processing of data from Warp. Also, it can sometimes perform memory access-intensive but not compute-intensive computations as well as or better than the Warp array, which can also allow the Warp array to be used for something else in the meantime.

ACKNOWLEDGMENTS

The Warp group, a large and growing group at Carnegie Mellon and General Electric, has contributed to this work by the design, implementation, and maintenance of the Warp machine and associated software. Research in robot locomotion cannot progress without reliable mobile robots, the design, implementation, and maintenance of which is a difficult problem in itself; we are therefore indebted to the Mobile Robots Lab at Carnegie Mellon for Neptune, and the Field Robotics Center at Carnegie Mellon for Terragator, both of which ran FIDO. We have also benefited from the research contributions of the Image Understanding Systems group at Carnegie Mellon.

REFERENCES

[1] Moravec, H. *Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover.* Technical Report CMU-RI-TR-3, Carnegie-Mellon University Robotics Institute, September, 1980.

[2] Thorpe, C.E. *FIDO: Vision and Navigation for a Robot Rover.* PhD thesis, Carnegie-Mellon University, December, 1984.

[3] L.H. Matthies, C.E. Thorpe. Experience with visual robot navigation. In *Proc. IEEE OCEANS'84 Conf.*, pages 594-7. IEEE, September, 1984.

[4] Kung, H.T. and Menzilcioglu, O. Warp: A Programmable Systolic Array Processor. In *Proceedings of SPIE Symposium, Vol. 495, Real-Time Signal Processing VII*, pages 130-136. Society of Photo-Optical Instrumentation Engineers, August, 1984.

[5] Annaratone, M., Arnould, E., Cohn, R., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O., Sarocky, K., Senko, J., and Webb, J. Warp Architecture: From Prototype to Production. In *Proceedings of the 1987 National Computer Conference*. AFIPS, 1987.

[6] M. Annaratone. *Warp Host Software Requirements and Deliverables* Carnegie Mellon Department of Computer Science, 1985.

[7] Gross, T. and Lam, M. Compilation for a High-performance Systolic Array. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*. ACM SigPlan, June, 1986.

[8] Bruegge, B., Chang, C., Cohn, R., Gross, T., Lam, M., Lieu, P., Noaman, A. and Yam, D. The Warp Programming Environment. In *Proceedings of the 1987 National Computer Conference*. AFIPS, 1987.

[9] T. Kanade and C. Thorpe with contributions from CMU SCVision Project Staff. *CMU Strategic Computing Vision Project Report: 1984 to 1985*. Technical Report CMU-RI-TR-86-2, Carnegie-Mellon University, The Robotics Institute, Pittsburgh, PA 15213, November, 1985.

[10] G. McApline, W. J. McLain, and G. B. Feldkamp. Controller smooths data flow through multiprocessor systems. *Electronic Design* :45-49, August, 1982.

[11] Dew, P. and Chang, C.H. Passive Navigation by a Robot on the CMU Warp Machine. Aug, 1984. Internal report, Department of Computer Science, Carnegie-Mellon University, Aug. 1984.

[12] Kung, H.T. Systolic Algorithms for the CMU Warp Processor. In *Proceedings of the Seventh International Conference on Pattern Recognition*, pages 570-577. International Association for Pattern Recognition, 1984.

[13] Kung, H. T. and Webb, J. A. Mapping Image Processing Operations onto a Linear Systolic Machine. *Distributed Computing* 1(4):246-257, 1986.

OPTICAL COMPONENTS FOR DIGITAL OPTICAL CIRCUITS

Brian S. WHERRETT, S. Desmond SMITH, Frank A.P. TOOLEY and Andrew C. WALKER

Department of Physics
Heriot-Watt University
Edinburgh EH14 4AS
Scotland, U.K.

The present status of all-optical computing for digital logic and information processing is reviewed briefly. Computing architectures that take advantage of the parallelism and interconnect freedom that optics offers are addressed. In particular demonstrator circuits employing optically bistable components are described.

1. CW OPTICAL BISTABILITY

Optical computing was first discussed seriously in the 1960's. In the context of the nonlinear optical devices available at that time, however, it was concluded that digital optical computational techniques could not hope to compete with electronics. Since then there have been great advances in electronic machines. Nevertheless the possibility of developing viable optical computing systems returned to prominence in the early 1980's. One key to the renewed interest in optical systems is the recognition that there are materials and devices for which one can mimic (and expand upon) the responses of electronic transistors, using optical beams at low power levels and employing small sized samples.

Figure 1 shows example transmission responses of a ZnSe-based narrow-band-pass interference filter, illuminated by the 514 nm wavelength radiation from an argon-ion laser [1]. Several features of this opto-thermal device response are significant. The response may be tailored by altering the cavity detuning (in this case by orientation) to either a hysteretic "optically bistable" character or to a highly nonlinear but non-hysteretic "transphasor" response with differential gain. The former provides a memory function, the latter leads to digital logic.

The simplest form of narrow-band interference filter consists of a spacer layer, of integral number of half waves optical thickness of a high index material, surrounded by quarter-wave stacks (alternating low/high refractive index), all deposited on a substrate (for example, glass, G).

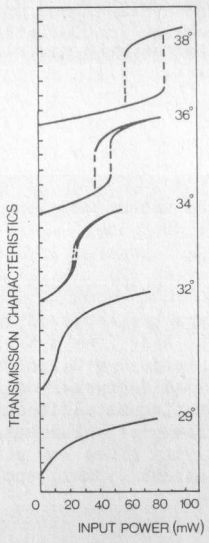


Figure 1.

We may thus describe the general design as G[m(HL)][n(HH)][m(LH)], in terms of quarter-wave optical thicknesses of high (H) or low (L) index material. A typical structure may contain about 17 layers. Our currently preferred materials are ThF_4 for the low-index layers (n = 1.4) and either ZnSe or ZnS for the high index (n = 2.5). These are deposited by thermal evaporation in vacuo or more recently by an ultra-high vacuum molecular beam technique. The total physical thickness of the multilayer structures is typically 0.5-5 μm.

The process responsible for optically bistable switching in these devices is as follows. The interference filter has a narrow band of transmission (∿ 4 nm full width at half maximum (FWHM)) centred just on the short-wavelength side of the 514 nm emission of an argon-ion laser. In the presence of laser radiation absorption in the spacer layer produces an increase of the temperature (similar devices exist in which electronic excitation by the direct absorption of radiation is responsible for the nonlinearity). The temperature rise causes an increase in the refractive index of the spacer material and hence in its optical path-length. The transmission peak therefore moves towards the operating wavelength. This in turn increases the intensity inside the spacer and can lead to an avalanching effect on the energy absorbed and the transmission. Thus the filter, under the correct initial conditions, will switch suddenly from low to high transmission as shown in Figure 1.

The output power levels, and in particular the differences between the outputs at switch-ON and switch-OFF of the bistable response, are sufficiently large that the system can be cascaded. The output from one device can be used to switch further devices, given the presence of bias beams close to the switch point but sufficiently far from it that noise-switching is avoided [2].

At present the power levels involved are cw (1-100 mW) and the interference filter operates at room temperature with a laser capable of 10 W or more of continuous output. The technology of filter fabrication is well established and good spatial uniformity across areas exceeding 1 cm^2 is achievable. In principle therefore a high degree of parallelism is achievable in a practical device.

The characteristic power levels and switching times for optical bistability of a thermal origin in a thin Fabry-Perot filter are [3,4]:

$$P_c \simeq \frac{\lambda_v \alpha \kappa_s r_o}{|\partial n/\partial T|} f' \quad ; \quad t_s \simeq \frac{\rho C_p r_o^2}{4\kappa_s}$$

λ_v is the vacuum wavelength, α the absorption coefficient of the Fabry-Perot spacer, κ_s the substrate conductivity, r_o the radiation spot radius. f' is of order unity for a cavity of optimised thickness (D) and reflectivity (R). $\partial n/\partial T$ is the spacer thermo-optic coefficient.

Low powers are achieved for high values of $|\partial n/\partial T|/\lambda_v \alpha$, which implies that operation away from a high absorption region is preferred if the cavity can be optimised, $D \simeq 2(1-R)/\alpha$. This has been demonstrated dramatically in recent liquid crystal cells, where non-absorbing nematic films in cavities incorporating metallic layers, have led to 140 μW and diode-laser-pumped bistability [5]. The equations also indicate that a low substrate conductivity gives low power switching, but at the penalty of an increased time constant. Small spot sizes are essential for both low power and high speed [6].

Typically filter switching has at present a time scale of order 10-50 μs. The majority of bistable responses of an electronic origin operate faster than the above but require high (pulsed) powers. One thus has the possibility of fast sequential logic or of slower, parallel devices. In this paper we concentrate on the latter for reasons that will be discussed. Table 1 indicates the parameters for present cw bistable systems. The degrees of

Table 1. Bit rates for optically bistable devices.

	Spot size	Separation (estimated)	Switch Time	Recovery Time	Power	Elements /cm²	Gate Hz/cm²
InSb [7]	20μm	30	200ns	200ns	2mW	5,000	8×10^9
ZnSe	5μm	20	10μs	50μs	.7mW	14,000	2×10^8
Liquid Crystal [5]	10μm	40	200μs	200μs	.2mW	50.000	1×10^8
SEED [8]	100μm	400	30ns	30ns	1mW	1000	2×10^{10}
ZnSe (predicted)	2μm	10-100μm	10-100ns	1μs	10μW	$10^4 \rightarrow 10^6$	$10^{10} \rightarrow 10^{12}$
Materials review [9]	5μm	10μm	5ns	5ns	100μW	10^6	10^{14}

parallelism achievable, based on a power dissipation of 10 W cm^{-2} and on the assumption that large arrays of devices could be fabricated, are given. Operation of large parallel arrays demands pixellation of both the active regions and of part of the substrate, in order to avoid transverse heat loss and crosstalk. Improvements of the individual device specifications indicate that, for either the ZnSe filter or for liquid crystal cells, a realistic attainable parallelism is of order 10^4-10^5 cm^{-2}. For the remainder of this paper we have in mind the future availability of such parallelism, combined with switch times of the order of microseconds, giving a computational power of 10^{10} Gate Hz cm^{-2}.

The normal disadvantages of thermal effects (slowness, crosstalk) are overcome by going to small pixels. Most optically bistable devices are influenced by the build up of temperature. The filter system therefore actually takes advantage of what is usually a deleterious effect, and in small volumes low switching energies ($\sim$ 10 pJ) are predicted. Recent picosecond-laser induced switching, occurring on a nanosecond timescale (but using relatively high powers), also demonstrates that thermal effects can be very fast [10].

Figure 2 shows example experimental nonlinear responses for (a) memory and (b) NAND-logic. In either case a bias holding beam of no information content is incident of the device. The figures indicate power levels for the hold and signal levels. In case (a), a temporary input signal causes the device output to alter from A (logic level-0) to B. On removal of the signal the bias beam maintains the output at C (logic level-1); the device latches.

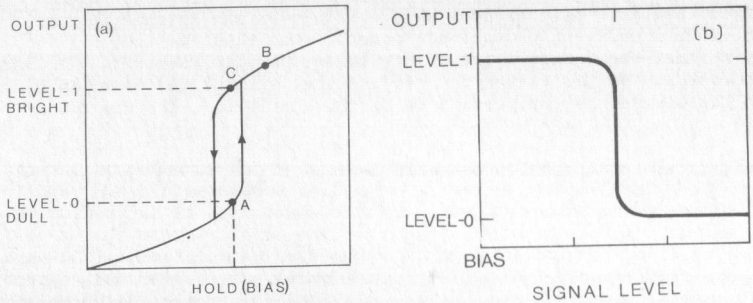


Figure 2. Optical latching and logic functions.

Switch off is achieved by removing the bias beam. Case (b) shows a device operated in reflection under non-hysteretic conditions. In the presence of the bias power and zero or one high "off-axis" signal level the reflected bias beam is at level-1. A second, simultaneous signal beam produces a level-0 output, thus providing a NAND-gate logic response.

2. PARALLELISM AND INTERCONNECTS

Over the past forty years it has been primarily the change in computer architectures that has led to the enormous increase in computation rates, particularly the introduction of parallelism, made possible by the reduction in size of individual gates and memory elements rather than by their operating speeds.

Thus if we consider the Goodyear 'Massively Parallel Processor' (MPP) with an array of approximately 10^4 processor elements, each linked to its four nearest neighbours and each with its own memory. Built for LANDSAT satellite data processing, limitations to the processing time are to a large extent set by input/output times and by the restriction of local interconnects within the array. Equally in more conventional machines the acquisition and communication of information off-chip is limited by the technical problem of hard-wiring multiple interconnections with no crosstalk. It is in the combination of parallelism and interconnect ability that optics would seem to give the greatest benefits, using macro- and micro-optics, holography and free-space propagation with no crosstalk. A hologram fabricated to give a single but arbitrary image pattern is in principle capable of making interconnects between 10^4 and 10^4 pixels. This freedom of interconnection, which is 2-D and can be non-local, contrasts with the nearest-neighbour coupling of the MPP and with present chip-to-chip restrictions of less than 300 links. The point is that in free space optical beams can be passed through each other with no crosstalk, in complete contrast to current carrying wires; it is the non-interacting quality of photons that makes them ideal for optical communications. It has been the requirement for phenomena that cause photons to interact strongly that has held up the progress of optical computing. The enhancement of radiation levels within narrow-band-pass filters and the sensitivity of these filters to refractive index changes of order 1 part in 10^3 is responsible for the potential use of low power optical bistability in all-optical switches and logic. One foresees use of optical 1-D and 2-D parallel interconnects both interfaced with electronic logic/memory and in all-optical architectures.

3. OPTICAL ARCHITECTURES

It is useful to discuss some of the optical architectures that have been suggested to date in terms of the amount of **FAN-OUT** or **FAN-IN** that they demand. This is because the number of logic gates that must be driven directly or indirectly from the output of single logic gate determines both the speed of processing and the tolerances on the gate and/or memory response characteristics.

3.1 Data Rearrangement

A number of shuffle networks are now being considered in electronic machines, to produce arbitrary information rearrangements in the most efficient ways. It has been suggested in particular that the perfect shuffle optical implementation will play a major role.

The perfect shuffle forms the basis of a fast sorting scheme. After each shuffle a decision is taken as to whether adjacent data items should be interchanged. The non-local nature of the shuffle reduces the number of steps required to achieve reordering; it is this nature that also gives optics the advantage over electronics. As a communication network, if used with electronic logic, no optical fan-out is demanded, although it is of course necessary to have optics-to-electronics interfacing and vice versa.

3.2 Image Processing

2-D image processing is the natural way for digital optical logic to be used. This may be the processing of a visual image: e.g. edge-extraction, noise-removal, joining of points; or it may be the processing of a binary image that models a physical system. Figure 3 shows an example using holographic fan-out/in combined with nonlinear logic plates. The scheme takes a black/white input image and produces the edges of any black regions as the output.

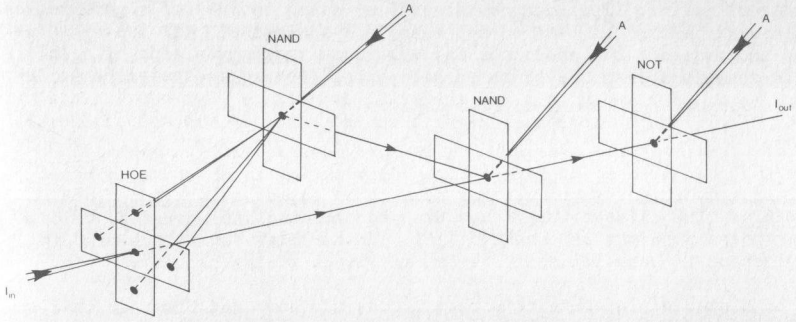


Figure 3. Elementary cellular logic all-optical image processor, using a pipelined architecture.

In a pipelined architecture, as depicted for the edge extractor, 2-D signal arrays are passed through a set of logic function plates. The necessity of using at least one logic plate for each operation limits the number of operations that could be employed and hence limits the flexibiilty of the computational device, but a high input rate is acceptable. Figure 3 shows the optical implementation of a binary, edge extraction process. "A" represents a beam from a holding array. I_{in}, I_{out} are elements of the input and processed images, HOE is a holographic element. We have constructed such a device, initially with only five information channels,

the point being to demonstrate the use of efficient holographic (dichromated gelatin) elements, used as both generators or uniform arrays of beams and as fan out/in devices. The use of nonlinear Fabry-Perot etalons for NAND and NOT operation by adjustment of holding beam levels is similarly demonstrated as is the cascadability of the signal levels.

Whilst a pipelined system can be programmed for various processes if sufficient numbers of logic plates are introduced it is still relatively inflexible. A complementary architecture is achieved with very few logic plates by iterating the processed information and altering the function of the plate/s at each circuit [11]. To prevent contamination between an input to a logic pate and the processed image that it produces, the circuit must contain a time delay longer than the gate switch time. For fast-pulse operation optical time-of-flight may be sufficient (33 ps per cm). For steady-state operation, where one is concerned with signal power levels, an artificial delay must be introduced. This corresponds to a temporary memory (write-read-erase), and is achievable by a clocked, bistable plate and a shutter, Figure 4.

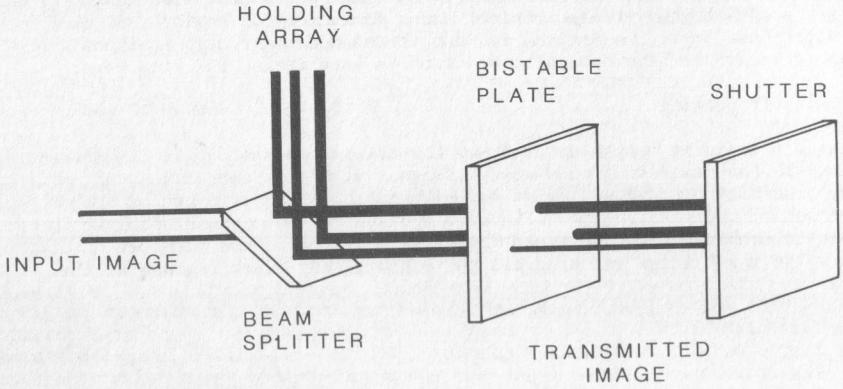


Figure 4. Optical 2-D memory or time-delay.

The use of three bistable plates in series provides a pair of shuttered memories that can be used to form a time-delay buffer in an otical circuit [11]. Such a synchronous circuit has come to be called a lock-and-clock architecture. The circuit itself has been constructed and operated, to date with just three information channels [12]. In parallel format the circuit will correspond to parallel shift registers with a wiring harness.

The use of bistable plates as temporary stores has an added bonus in that the output levels are both standardised and can be set to precisely the signal levels requried of the processing, logic gates, thereby avoiding error accumulation. In the lock-and-clock circuit each plate was identical, a high signal (level-1) could be shunted around the circuit without alteration, demonstrating that restoring logic is achievable optically. Similar circuits have also been operated as inverters and optical flip-flops (Figure 5).

Schemes for implementing an optical 'Cellular Logic Image Processor' [14] and a parallel Full-Adder [15] have been devised. Individual transphasor plates demonstrate OR,AND; NOR,NAND functions. Combinations of two plates have demonstrated XOR,XNOR action to date. Full-addition at a single pixel, utilising both transmission and reflection responses, has also been demonstrated in a simplified circuit.

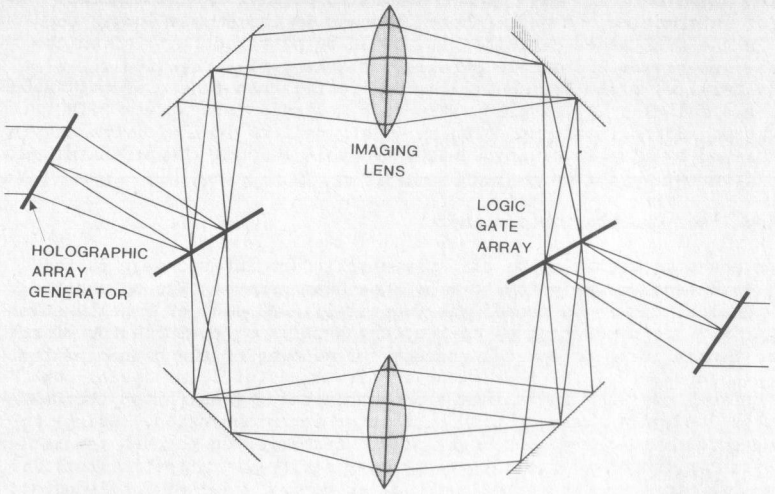


Figure 5. Multichannel all-optical flip-flop. The reflected holding beams from each logic gate array act as off-axis input signals for the other gate array. The circuit acts in a Set-Reset mode [13], using external signals onto individual channels.

Figure 6 shows the full-adder truth table and its optical implementation. Discrimination at the levels indicated is achieved by using the transmitted/reflected beams as off-axis addresses to nonlinear plates. Figure 2 showed an example of the hard-limiting achieved when a reflected holding beam is switched by a small signal incident at an angle [16]. By incorporating a full-adder plate in a synchronous circuit similar to that described above the discriminated signals can be used as further inputs for addition. In this way either word-parallel additions and integrations or bit-slice additions of planes of data can be achieved. A 10^4-element plate would achieve 10^4 full single-bit additions in each cycle time. Once again the demonstration circuit has been used as an existence-proof.

A	B	C	S	C'
0	0	0	0	0
0	0	1	1	0
0	1	1	0	1
1	1	1	1	1

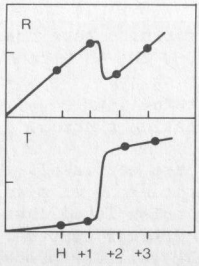


Figure 6. Truth-table for full addition and use of simultaneous Reflection and Transmission signals at a single nonlinear pixel, to achieve the truth-table. Horizontal dashes indicate required discrimination levels.

Whilst the above circuits have been operated to date with only limited numbers of information channels (three channels in the simple loops, one channel in the full-adder circuit), it should be stressed that, given the beam array generation and sample pixellation, the optics required for a massively parallel plate device will be no greater than that presently used. One then has a lock-step, single instruction multiple data stream "SIMD" architecture. Both circuit synchronism (clocking) and logic programming is to be achieved by bias level adjustment. In this respect the programming is separate from the data flow.

3.3 Generalised Crossbar Interconnect

This is a powerful architecture that gives arbitrary interconnect patterns (1-to-1, 1-to-many, or many-to-1), between linear arrays. The output from one of N optical pixels is fanned out (vertically) to each of N cells on an N x N matrix. The throughput of each of the N cells on a single row of the matrix is fanned in to an optical element. Depending on the nature of the optical elements and/or the matrix the architecture fulfills several uses. A 32 x 32 optical crossbar switching is presently being constructed for DARPA and a 256 x 256 system is sought, for computer interconnection. Using source/detector systems and analogue optical transmission through the matrix this device has been used for optical vector-matrix multiplication and is being employed to investigate the associative memory model of Hopfield.

3.4 Global Interconnects

The use of a lens as a Fourier transform in 2-D beween object and image planes is the classic example of processing using global interconnection and has of course led to a wide range of applications. The combination of Fourier lensing with Fourier plane correlation by four-wave-mixing is an example of image processing using nonlinear optics, that has considerable device applications. The combination of Fourier plane techniques, holography and optical thresholding is also being used now for optical pattern recognition by associative memory [17].

4. CONCLUSIONS

Three years ago digital optical circuitry was at the stage of the silicon technology of 1958, at which time Jack Kilby of Texas Instruments coupled together two gates on a single silicon chip. By 1984 two bistable devices on a single **optical chip** of InSb had been coupled to produce an XOR-gate. Using ZnSe and ZnS optically bistable interference filters prototype all-optical pipelines and circuits have now been constructed and arrays of bistable elements, up to 10 x 10, have been operated simultaneously. These and other demonstrators, using hybrid spatial light modulators, holograms, etc. are essential to the establishment of specifications for optical components in digital processing systems.

Present nonlinear elements are not ideal. Technological advances in fabrication and thermal engineering of pixellated devices and the generation of high uniformity and low noise-level laser beam-arrays are priorities, even for modest demonstrators. Optical computer architectural ideas appear to be homing in on a 2-D parallel, cellular processor scheme, with various interconnect configurations. One must take advantage of <u>both</u> the parallelism achievable optically and the interconnect freedom. In addition the parallel data acquisition from optical (disc) storage is seen as essential to the advancement of optical systems. Even so the development of a general purpose optical computer does not seem likely without a dramatic breakthrough in the fundamental physics of nonlinear phenomena. However, given a fraction of the invested effort in microelectronic integrated circuitry over the past forty

years, it would be surprising if specialist digital optical devices of value were not constructed before the turn of the century.

ACKNOWLEDGEMENTS

The efforts over a period of years of many members of the Heriot-Watt Optoelectronic Research Group is acknowledged. The Science and Engineering Research Council are recognised for funding support.

REFERENCES

[1] Smith, S.D., Mathew, J.G.H., Taghizadeh, M.R., Walker, A.C., Wherrett, B.S. and Hendry, A., Opt. Commun., 51, 357 (1984).
[2] Smith, S.D., Walker, A.C., Wherrett, B.S., Tooley, F.A.P., Mathew, J.G.H., Taghizadeh, M.R. and Janossy, I., Appl. Opt., 25, 1586 (1986).
[3] Wherrett, B.S., Hutchings, D. and Russell, D., J. Opt. Soc. Am. B3, 351 (1986).
[4] Wherrett, B.S., in 'From Optical Bistability Towards Optical Computing', Ed. P. Mandel, S.D. Smith and B.S. Wherrett, Elsevier, in press.
[5] Lloyd, A.D., Janossy, I., MacKenzie, H.A. and Wherrett, B.S., Opt. Commun., in press.
[6] Mathew, J.G.H., Taghizadeh, M.R., Abraham, E., Janossy, I. and Smith, S.D., Springer Proc. Phys., 8, 57 (1986).
[7] Al-Attar, H.A., Hutchings, D., Russell, D., Walker, A.C., MacKenzie, H.A. and Wherrett, B.S., Opt. Commun., 58, 433 (1986).
[8] Miller, D.A.B., Chemla, D.S., Damen, T.C., Wood, T.H., Burrus, C.A., Gossard, A.C. and Wiegmann, W., IEEE J. Quantum Electron., QE-21, 1462 (1985).
[9] Auston D. et al., Appl. Opt., 26, 211 (1987).
[10] Chow, Y.T., McGuckin, B.T., Smith, S.D., Wherrett, B.S. and Van Stryland, E.W., CLEO Tech. Digest Paper ThK36 (1987).
[11] Wherrett, B.S., Appl. Opt., 24, 2876 (1985).
[12] Smith, S.D., Walker, A.C., Tooley, F.A.P. and Wherrett, B.S., Nature, 325 (19897).
[13] Wherrett, B.S., IEEE J. Quant. Electron., QE-20, 646 (1984).
[14] Wherrett, B.S., in Proc. Workshop on Optical Bistability in Optical Computing, Huntsville (1985), in press.
[15] Wherrett, B.S., Opt. Commun., 56, 87 (1985).
[16] Walker, A.C., Appl. Opt., 59 (1986).
[17] Abu-Mostafa, Y.S. and Psaltis, D., Sc. Am., 256, 66 (1987).

A REDUCTION ARCHITECTURE FOR THE OPTIMAL SCHEDULING OF BINARY TREES

K. Ravikanth, P.S. Sastry, K.R. Ramakrishnan and Y.V. Venkatesh

Department of Electrical Engineering
Indian Institute of Science
Bangalore, India

This paper addresses the problem of designing a parallel reduction architecture for applicative languages. An interconnection network that allows for scheduling of binary trees of arbitrary depth is presented. It is shown that using a static scheduling strategy the architecture achieves optimal performance while scheduling complete binary trees. Some issues related to the design of a machine based on this network are also discussed.

1. INTRODUCTION

Over the years it has become increasingly evident that future solutions for high speed computing cannot depend on technology alone. This realization has led to a restructuring of the notion of a computer with the concept of parallelism emerging as the key to building faster machines. Thus, the design of parallel computers for either a special class of problems or general purpose applications has been the main thrust of research in architecture. The advent of VLSI technology has given these efforts a basis for practical realization.

For programming these parallel machines, many of the existing languages have, in fact, been extended to suit the parallel framework. But these efforts have not resulted in elegant parallel programming methods as these languages are closely tied to the von Neumann computer model in their semantics. Backus [1] was apparently the first to realize this and he proposed the so called 'applicative semantic model' as a basis for future language design as an alternative.

In the applicative semantic model, as is well known, a program unfolds recursively into subtasks giving it a tree structure. It follows from the Church-Rosser property [2] of these languages that the final result of the program is independent of the order in which these subtasks are evaluated. This property makes the applicative languages ideally suited to the domain of parallel computing since neither special techniques for extracting parallelism on the part of the compiler nor special efforts to express parallelism on the part of the programmer are required.

By virtue of this property of the applicative semantic model, several machines called reduction machines, have been proposed [10]. The main contribution of the present paper is the description of an interconnection network which may be employed to construct a reduction machine that efficiently executes applicative languages in parallel.

We first present an interconnection network which turns out to be a binary de Bruijn graph [9]. A static scheduling policy is presented using which equal load can be maintained on each processor when a complete binary tree task structure is distributed on this network. This is realized along with minimum possible interprocessor communication. We also present some results regarding the performance of the network under static scheduling for

task structures which are not complete binary trees. Next, we explore certain issues related to the design of a machine based on the interconnection network presented. At the end, we make a comprehensive comparison of our work with related efforts in this area.

2. THE INTERCONNECTION NETWORK

Architecture design addresses itself to the task of configuring a physical structure that best meets the problem requirements. Since performance of a parallel architecture can be charecterized mainly by communication delays, distribution of load among processors and scheduling overheads, a close correspondence between the structure of the problem and the architecture is desired in order to minimize these overheads. Thus, in the context of parallel architectures, this essentially implies the need for the right choice of an interconnection network and a scheduling strategy which together can achieve the mapping of the problem on to the machine so as to minimize these overheads. When the problem structure is such that it recursively decomposes itself into identical subproblems, the architectural requirements can be formulated as follows:

> Given a finite set of identical processors, what interconnection network and scheduling strategy give rise to a virtual tree structure of arbitrary depth ? (*)

Before presenting the proposed interconnection network and the corresponding scheduling strategy, we identify certain ideal properties desired of any parallel architecture to be satisfied. An architecture with these properties minimizes overheads thus guaranteeing maximum possible speedup.

Properties for ideal peformance:

P1) Minimum distance property: This property assures the minimization of communication in distributing subtasks and collecting partial results. This property may be formally stated as: If T and T' are two tasks from a task tree of a given problem such that T is the parent of T' and if P and P' are the processors T and T' are scheduled on respectively, then P should be directly connected to P' in the network.

P2) Static Scheduling: When a task spawns subtasks, there should be a simple static rule to decide which processors these subtasks may be executed on. This is desirable since it minimizes scheduling overheads and obviates the need for processors to exchange status information.

P3) Minimum load imbalance: For a task structure that recursively unfolds as a tree, computation is active only at a single level - the level that is currently the lowest one. Since the tasks that are computationally active at any instant come from the same level, it is desirable that the tasks at every level of the task tree are distributed evenly among all processors in the network. This property may be formally stated as: If P' and P" are any two processors belonging to the network, then $|load_k(P') - load_k(P")| \leq 1$ (where $load_k(P)$ denotes the number of tasks belonging to level k of the task-hierarchy that are scheduled to execute on P).

A network structure and scheduling strategy that can satisfy properties P1 - P3 for a given class of problems give us the best possible solution for the problem (*). It is not always possible to arrive at such a solution for every class of problems. For example, consider the problems that assume the structure of an arbitrary tree. It is impossible to configure a network that can satisfy properties P1 - P3. However, if we restrict ourselves to complete trees, it is possible to find solutions that satisfy all the three properties.

In what follows, we first present a network structure and a scheduling strategy satisfying properties P1 - P3 for task structures which assume the form of a complete binary tree and then generalize them for complete k-ary trees.

2.1 The Interconnection Scheme for Binary Trees

The interconnection network we present turns out be a binary de Bruijn graph [9]. It has the property of a binary tree folding onto itself giving the appearence of infinite depth.

Let Q be a set of N identical processors. We can then define the interconnection network in terms of two functions L and R, which map Q onto Q, as follows:

$$\text{If } Q = \{P_0, P_1, \ldots, P_{N-1}\} \text{ then,}$$

$$L(P_i) = P_{2i \bmod N} \text{ and}$$

$$R(P_i) = P_{(2i+1) \bmod N}, \text{ for all } P_i \text{ in } Q. \qquad (I)$$

The functions L and R establish links from the processor P to the processors L(P) and R(P), for each processor P in Q. It turns out, as a result of the above defintions, that each processor has a connectivity of four in this network. Fig. 1 shows, for instance, the interconnection network for eight processors.

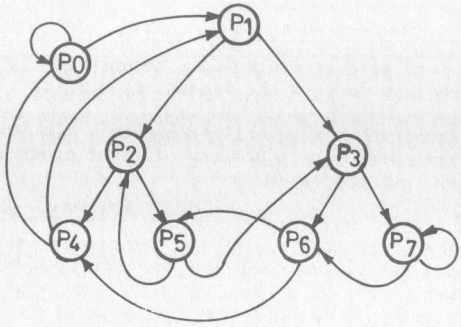


FIGURE 1
The Interconnection Network for Eight Processors

The functions L and R are equivalent to the left-child and right-child of a processor respectively for any processor. By composing the functions L and R, we can refer to processors that are not directly connected to a processor. For example, LRR(P) would refer to the left-child of the right-child of the right-child of P. By associating a direction with the arcs from P to L(P) and R(P), as shown in Fig. 1, we can evolve a simple static scheduling strategy which is described below:

> Scheduling strategy: Let a task T on a processor P from Q spawn two subtasks T' and T" then, a simple static scheduling amounts to assigning T' to processor L(P) and T" to processor R(P). (II)

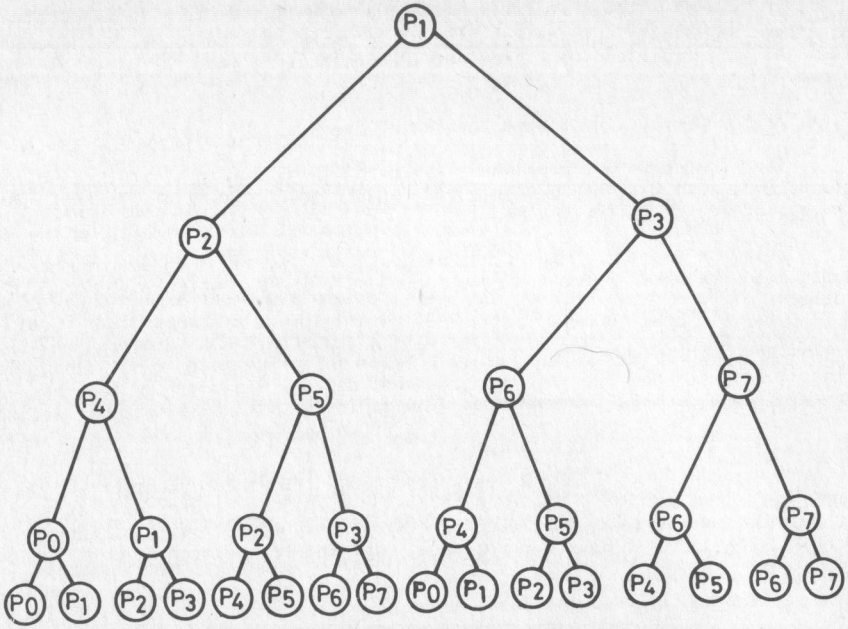


FIGURE 2
An oversized problem distributed on a network of eight processors. Marked on each task node is the processor to which it is assigned.

Fig. 2 illustrates how an oversized problem, a complete binary tree with 31 tasks, gets distributed onto a network of eight processors using the static scheduling mechanism just described.

We now demonstrate that complete binary trees can be scheduled on this network rooted at any processor, satisfying the properties P1 - P3. P1 follows from the fact that every task downloads the two subtasks it spawns on its immediate neighbours. P2 is evident from the fact that scheduling is done using the strategy (II). P3 follows from the theorem given below:

Theorem: The scheduling strategy (II) distributes tasks at every level of the complete binary tree in a round robin fashion among the processors in the network (I).

Proof: We shall prove by induction, for any level k of the task tree, that the tasks at that level are distributed in a round robin fashion among the processors in the network. Let us assume that the task at the root node is scheduled on processor P, for some P in Q. For notational simplicity we shall henceforth refer to a processor by its index.

Since there is only one task at level zero, it follows trivially that tasks at this level are distributed in a round robin fashion. Now, assume that the tasks at level k-1 of the task-hierarchy are distributed in a round robin fashion. Let j and (j + 1) mod N be two consecutive processors in Q on which two consecutive tasks at level k-1 have been scheduled. The tasks at level k spawned by these two tasks, according to the scheduling strategy, are assigned to the following processors:

$2j \bmod N$, $(2j + 1) \bmod N$, $(2j + 2) \bmod N$, $(2j + 3) \bmod N$

From this it follows that tasks at level k are assigned in a round robin fashion since two consecutive tasks at level k are assigned to consecutive processors. #

Remark: The scheduling strategy adopted distributes tasks at every level of the complete binary tree in a round robin fashion among the processors in the network, with the left most task at the k-th level being assigned to the processor $L^k(P)$ (P being the processor to which the task at the root is assigned) . This implies that no two processors in the network can have a difference of load which is greater than one, thus leading to the minimum load imbalance property P3.

2.2. The Interconnection Scheme for k-ary Trees

Though the interconnection network and scheduling strategy are specific to the case of the complete binary tree, they can be generalized to optimally schedule complete k-ary trees. The interconnection network can be understood interms of k functions which map Q onto Q. The j-th function of these, for $1 \leq j \leq k$, is defined as:

$$Child_j(P_i) = P_{(k*i + j-1) \bmod N} \quad , \text{ for every } P_i \text{ in Q.}$$

Every processor in Q has a connectivity of 2*k in this network. A static scheduling policy, on similar lines to the one described for the complete binary tree case can be evolved. According to this policy when a task on processor P spawns k subtasks, they get assigned to the following processors:

$$Child_1(P) \ , \ Child_2(P) \ , \ \ldots \ , \ Child_k(P)$$

A complete k-ary task tree can be distributed on the network described above using the static scheduling policy presented. Such a distribution is optimal since it can be shown, proceeding along the lines taken to prove for the case of the binary tree, that the properties P1 - P3 are satisfied.

3. SOME ISSUES RELATED TO MACHINE DESIGN

The interconnection network presented in the previous section is the basis of our architecture which, as a consequence of the theorem, is ideal for evaluating complete binary tree structures. In this section we present some results regarding the performance of the network on a larger class of tree structured problems. We also present a method of reducing communication overheads in our architecture through the use of shared memories.

3.1. Expected Speedup

We derive an expected speedup for complete binary tree task- hierarchies under the assumption that no structured data is passed between the processors. The task tree is assumed to unfold to a depth of n .

For a task tree that unfolds recursively, each task has associated with it a distribution time tD and an execution time tE . The time taken to execute the task tree then would be:

$$T = (tD + tR) * (load_0 + load_1 + \ldots + load_n)$$

Load on a processor due to the k-th level of the task tree is given by

$load_k = \lceil 2^k / 2^m \rceil$, where N is assumed to be 2^m .

Thus, it follows that:

$$T = (tD + tE)*(n + 1) \qquad \text{for } n < m$$

$$T = (tD + tE)*(m + 2^{n-m+1} - 1) \qquad \text{for } n \geq m$$

The distribution and execution steps exist even when the task tree is executed on a uniprocessor machine. The distribution step, in this case, corresponds to the creation of an activation record on a function call. Let tD' and tE' be the respective times taken for the distribution and execution steps. The time taken to execute the task tree in the uniprocessor case then would be:

$$T' = (tD' + tE')*(2^{n+1} - 1)$$

Since tD will be fractionally greater than tD' in our architecture as the extra information a task packet has to contain is only that regarding the destination processor and tE is equal to tE', we get a speedup solely in terms n and m as follows:

$$T'/T \approx (2^{n+1} - 1)/(n + 1) \qquad \text{if } n < m$$

$$T'/T \approx (2^{n+1} - 1)/(m + 2^{n-m+1} - 1) \qquad \text{if } n \geq m$$

The speedup achieved is always greater than one and becomes asymptotic to N as n grows large when compared to m, that is, when the problem wraps around the network several times. Though this speedup is ideal, we expect the actual speedup to be quite close to this for problems that do not handle structured data since communication between processors can be made efficient by the use of shared memories as decribed later in this section.

3.2. Scheduling Arbitrary Tree Structures

Since our network is ideally suited for scheduling complete binary trees, a valid question that has to be answered is: " What is the basis of the choice of a binary tree, as an abstraction for the dynamic structure of an applicative program ?"

Recursive algorithms having exploitable parallelism generally unfold as binary trees since they have a 'divide and conquer' structure. Thus we belive, an architecture designed to efficiently execute binary trees is on a sound footing. This belief has been strengthened by the simulation studies conducted to investigate the performance of the network for other tree structures as well. These studies, though incomplete, indicate that the network has a tendency to smoothen out any imbalances in load among individual processors as the task tree wraps around it. The task structures considered for this investigation include cases of incomplete binary and ternary trees and the complete ternary tree.

An incomplete task tree is built using two random variables - one to decide whether a task should spawn subtasks or not, and another to decide the number of subtasks that should be spawned. By associating different probability distributions with these two random variables different kinds of trees can be generated.

When a task on a processor P spawns subtasks, these subtasks are assigned alternately to the processors L(P) and R(P), the left-child and the right-child respectively of P . Such a scheduling mechanism is simple but it does not satisfy the condition P3. The effectiveness of this scheduling mechanism can be understood by studying how the maximum load on a processor deviates from the

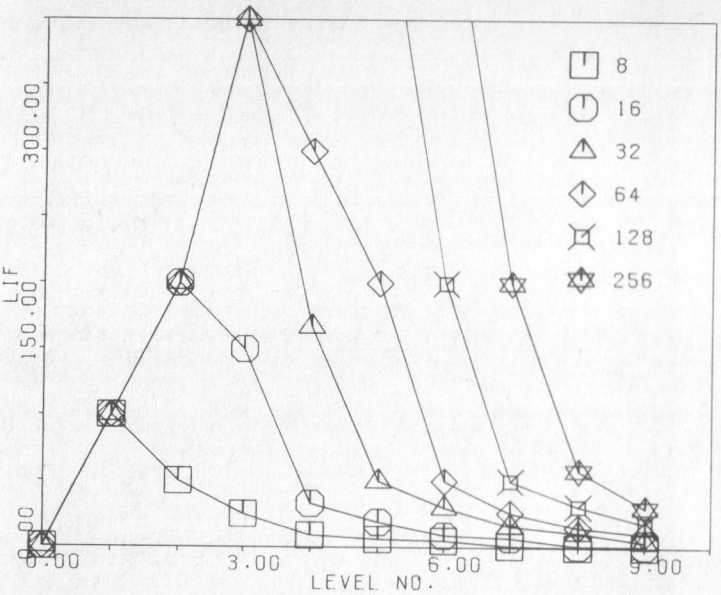


FIGURE 3 (a)

lif vs level no. for the complete ternary tree on the network sizes 8 - 256. lif peaks at (4 , 700) and (5 , 1100) (occuring outside the figure) for the network sizes 128 and 256 respectively.

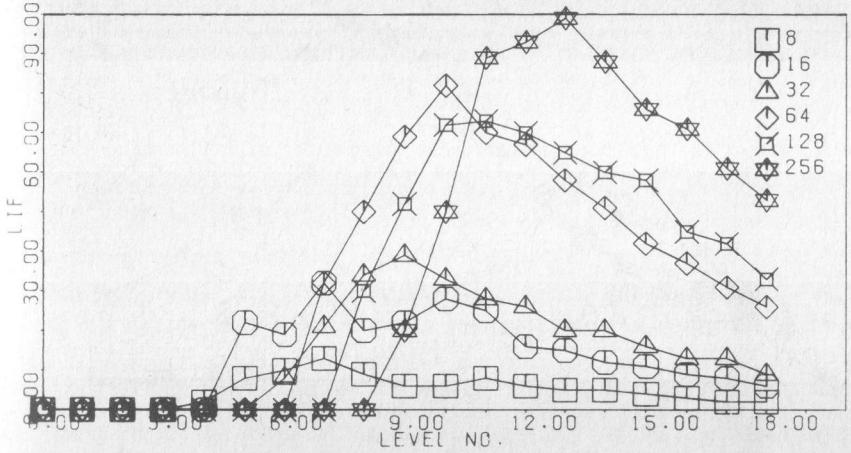


FIGURE 3 (b)

Average lif vs level no. for a class of incomplete binary trees on the network sizes 8 - 256. The trees were generated using the following parameters: Probability of spawning subtasks at any level = 0.75 and No. of subtasks spawned = 2

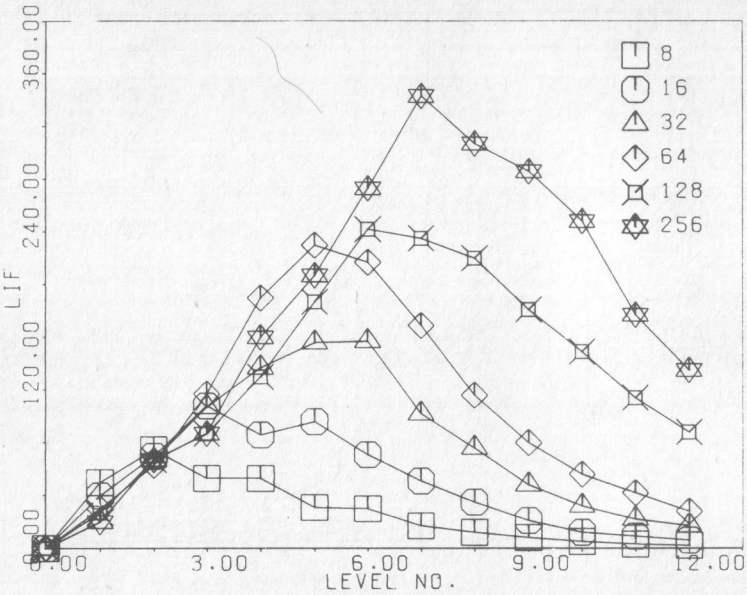


FIGURE 3 (c)

Average lif vs level no. for a class of incomplete ternary trees on the network sizes 8 - 256. The trees were generated using the following parameters: Probability of spawning subtasks at any level = 1.0 and No. of subtasks spawned are one, two or three with equal probability.

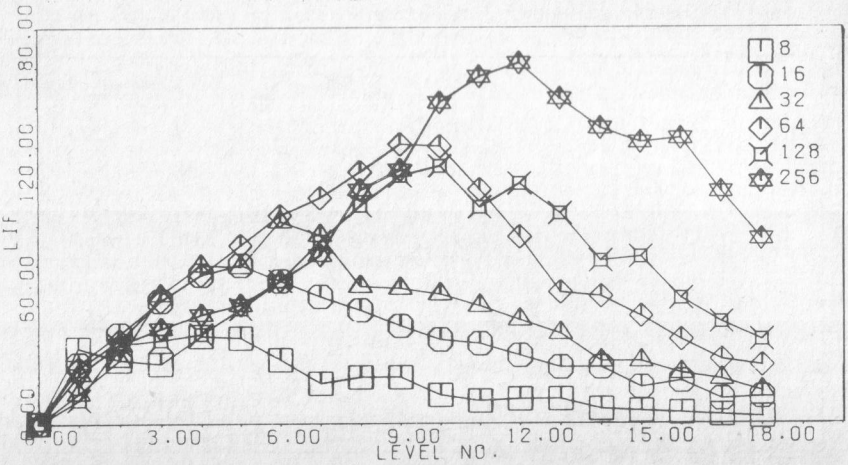


FIGURE 3 (d)

Average lif vs level no. for a class of incomplete ternary trees on the network sizes 8 - 256. The trees were generated using the following parameters: Probability of spawning subtasks at any level = 0.75 and No. of subtasks spawned are one, two or three with equal probability.

ideal load, for every level of the task tree.

To this end we introduce the term 'load-imbalance-factor' (lif) which is defined for every level of the task tree. It represents percentage deviation of the maximum load on a processor from the ideal load. The load-imbalance-factor for the k-th level of the task tree is defined as:

$$lif_k = 100*(\max_i \{load_k(P_i)\} - ideal\text{-}load_k) / ideal\text{-}load_k$$

$$\text{where } ideal\text{-}load_k = (load_k(P_0) + load_k(P_1) + \ldots + load_k(P_{N-1})) / N .$$

Here, $load_k(P)$ stands for the load on processor P due to the k-th level of the task tree.

The simulation studies conducted on the tree structures mentioned above indicate that the load-imbalance-factor approaches zero asymptotically as one goes down the levels of the task tree. When the size of the task tree is relatively larger than the size of the network, it has been observed that the load-imbalance-factor approaches zero much faster.

Figures 3 (a - d) represent the average behaviour of the load-imbalance-factor with respect to the level in the task tree for different randomly generated tree structures on various network sizes. It has been observed that the load-imbalance-factor shows a similar behaviour in all these cases, rising initially from zero to a peak and then reducing to zero asymptotically. The number of tasks, in those levels of the graphs where the load-imbalance-factor shows a rising trend, is less than the number of processors in the network. The load-imbalance-factor starts falling in these graphs once sufficient number of tasks are available in the network. It is also observed that for problems having a high degree of parallelism the load-imbalance-factor approaches zero quickly. This may be confirmed from the differences in behaviour the graphs for the complete and incomplete ternary trees exhibit.

A dynamic load balancing scheme may be employed to reduce differences of load between processors which arise as a consequence of the static scheduling mechanism. Each processor looks at the load on its neighbours and downloads a task to a neighbour whenever the neighbour has a lesser load, thereby amounting to a distributed load diffusion process . The load diffusion process and the task execution process may be realized either on two different hardware units in the processor or may be multiplexed on the same hardware unit. From the available data, it is difficult to say conclusively whether such a dynamic scheduling scheme is necessary or not. It is conjectured that the extra complexity involved in implementing the dynamic scheduling scheme may not justify the increment in speedup that may be achieved through it. This aspect is currently being investigated.

3.3. Use of Shared Memories

In a multiprocessor system two steps are involved in the distribution of tasks. They are, first, the creation of a task packet, and next, the communication of this task packet over the network to the destination processor. This time required for distribution can be further reduced through the use of shared memories between neighbouring processors in our network. The use of shared memories is possible only because every new task that is created on a processor is scheduled to execute on an immediate neighbour. If such an organization is employed, when a subtask is spawned, the task packet may be directly created in the memory of the destination processor.

Since each processor in the network has four neighbours, two connected to its incoming arcs and two to its outgoing arcs, it may appear at first sight

that the contention problem would make the choice of shared memories undesirable. This problem however, can be minimized by dividing the memory of each processor into three banks: left-memory, right-memory and local-memory. Here, the bank local-memory is local to each processor and contains all programs, relevant tables, etc. including a copy of the interpreter. Processors communicate through the memory banks they share. Fig. 4 shows the interconnection of neighbouring processors through the memory banks they share.

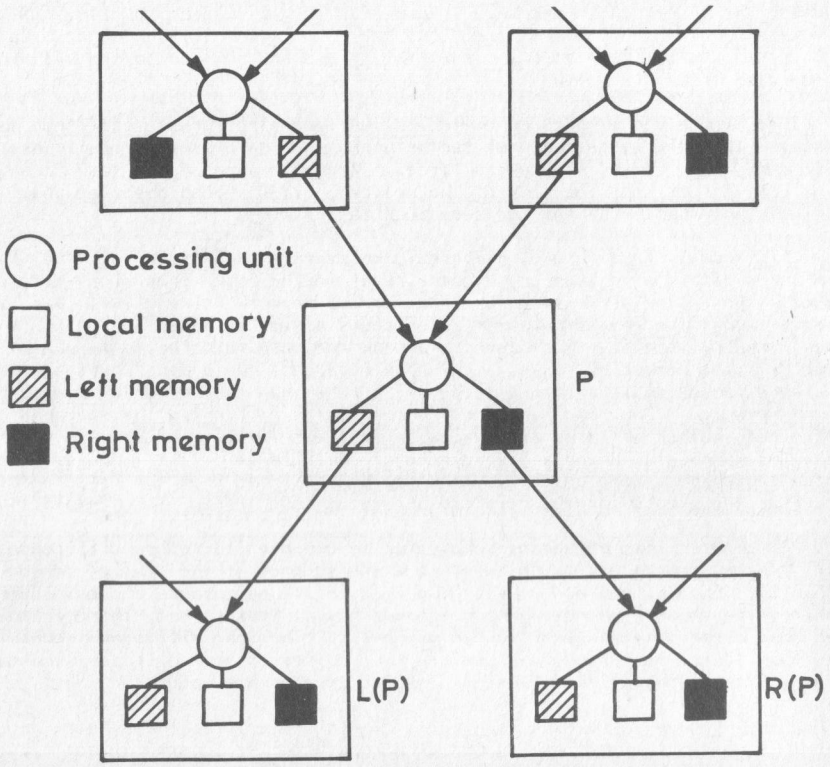


FIGURE 4
Shared memory interconnetion between neighbouring processors

When a task running on a processor P spawns two subtasks, the task packets for these subtasks are created in the left-memory and right-memory banks of the processor and are later directly accessed by the processors L(P) and R(P). Results of computation are also returned from the processors L(P) and R(P) to the processor P in a similar manner. When a processor's memory is divided in this manner, no contention arises amongst the four neighbours of a processor. Contention arises only when two processors sharing a common memory make a simultaneous access on it. However, since each processor divides its attention between five memory units and every shared memory bank can be accessed by only two processors, this contention is not significant.

4. COMPARISONS WITH RELATED WORK

Minimizing interprocessor communication and maintaining an even distribution of load among processors are primary goals in any multiprocessor design which are generally realized by judicious choice of an interconnection network and an efficient scheduling mechanism.

For programs that unfold as trees, we have shown that by using the interconnection network presented in this paper we are able to achieve a fairly good load distribution, along with minimal communication and no scheduling overheads. Though applicative programs having sufficient parallelism do unfold as tree structures, this property is not restricted to applicative programs alone but holds for any program written in a 'divide and conquer' fashion, irrespective of the underlying programming paradigm. Since our work has been motivated by the desire to configure architectural elements suitable to applicative languages, we compare our work mainly with other similar efforts.

Amongst parallel reduction machines Mago's cellular tree architecture [3] that executes Backus' FP is one of the pioneering works. The program is present in the leaf cells (occupying one symbol per cell) and the execution takes place in two phases. First comes an upsweep in which all reducible expressions are recognized, which is followed by a downsweep in which these expressions are rewritten by their values. In our architecture, unlike in Mago's machine, compututation first unfolds by building a tree structure in a top-down fashion followed by the reduction of the tree in a bottom-up manner.

AMPS is another tree structured machine which was proposed by Keller et.al. [6] . In this machine the processors at the leaves perform computation and the processors at intermediate node positions perform routing and load balancing. Maintaining a uniform load in the leaf processors leads to an increase in communication which can get severely affected by the bottleneck at the root.

The REDIFLOW project [5] was a follow up of AMPS which incorporated a distributed load balancing scheme. In this scheme diffusion of load takes place along "pressure gradients" existing between neighbouring processors. The "pressure" at a processor is computed as a function of load on the processor, available memory, etc. Simulation results indicate that a speedup of upto 30 can be obtained with programs that use the "divide and conquer" principle on a system consisting of 128 processors. In our architecture, with a network of 128 processors alone in size, the same speedup can be obtained on a problem that unfolds to eight levels. We get increasingly better results as the problem size grows larger. In addition, unlike in REDIFLOW, the scheduling is static in our case.

The Zero Assignment Parallel Processor ZAPP, designed by Sleep and Burton [8], comes closest to our architecture in performance. The designers have chosen an indirect binary n-cube as the interconnection network in their machine. A static scheduling algorithm is employed to distribute a binary tree task hierarchy on this network. This scheme spreads the nodes of the binary tree fairly uniformly among the processors in the network but fails to achieve optimal distribution as in our network.

There have been other efforts, though unconnected to reduction architectures, to configure interconnection networks to execute binary tree task structures. Martin [4] proposed the twisted torus, which is a modification of the hypertorus. Sequin [7] further modified it to obtain the doubly twisted torus. These networks, through a static scheduling strategy, distribute binary trees assigning tasks at level k of the tree unevenly among k+1 processors. Since the number of tasks at a level are related exponentially to the level number, these networks do not achieve optimal performance.

5. CONCLUSIONS

In this paper we have presented an architecture for implementing applicative languages efficiently. Our architecture minimizes communication overheads by the choice of an interconnection network which closely matches the problem structure. This network turns out to be a binary de Bruijn graph. For a class of applicative programs that unfold as complete binary trees, we are able to show that the network achieves an ideal performance using a simple static scheduling mechanism. This class of problems is significant because many 'divide and conquer' algorithms would result in such binary tree structures. However, for problems that assume the form of arbitrary tree structures, the network is able to achieve a fairly good distribution of tasks using static scheduling. In fact, it is seen that the performance of the network improves as the problem size grows larger. It is shown that by use of shared memories between neighbouring processors one can reduce the interprocessor communication. Whether or not dynamic scheduling can be employed to improve the performance in a significant manner is currently being investigated.

REFERENCES

1. Backus J., Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs, Comm. ACM, Vol. 21, Aug 1978, pp. 613-641.
2. Barendregt H.P., The Lambda Calculus: Its Syntax and Semantics, North-Holland, 1981.
3. Mago G.A., A Cellular Computer Architecture for Functional Programming, Proc. IEEE COMPCON, 1980, pp. 179-187.
4. Martin A.J., The TORUS: An Excercise in Constructing a Processor Surface, Proc. of the VLSI Conf. 1981.
5. Keller R.M., Lin F.C.H., Tanaka J., REDIFLOW Multiprocessing, Proc. IEEE COMPCON, Feb. 1984, pp. 410-417.
6. Keller R.M., Lindstorm G., Patil S., A Loosely-Coupled Applicative Multi-Processing System, Proc. AFIPS Nat. Comp. Conf., 1979, Vol. 48, pp. 613-622.
7. Sequin C.H., Doubly Twisted Torus networks for VLSI Processor arrays, Eigth Annual Symp. on Comp. Arch., pp. 471-480.
8. Sleep M.R., Burton F.W., Towards a Zero Assignment Parallel Processor, Second Int. Conf. on Dist. Comp. Systems, 1981, pp. 80-85.
9. Uhr L., Algorithm-Structured Computer Arrays and Networks, Academic Press, 1984.
10. Vegdahl S.R., A Survey of Proposed Architectures for the Execution of Functional Languages, IEEE Trans. on Computers, Vol. C-33, No.12, Dec 1984, pp. 1050-1071.

THE DUTCH PARALLEL REDUCTION MACHINE

H.P. BARENDREGT, M.C.J.D. VAN EEKELEN AND M.J. PLASMEIJER,

Computer Science Department
University of Nijmegen
Nijmegen, The Netherlands

P.H. HARTEL, L.O. HERTZBERGER AND W.G. VREE,

Computer Science Department
University of Amsterdam
Amsterdam, The Netherlands

In November 1984 three research groups at the universities of Amsterdam, Nijmegen and Utrecht started a cooperative project sponsored by the Dutch Ministry of Science and Education (Science Council). The first phase lasting until the end of 1987 is a pilot study and has as aim to answer the following question. Is it *possible and realistic* to construct an efficient parallel reduction machine? The present paper gives an outline of the problems concerning parallel reduction machines and of our research towards their solutions.

1. Introduction

The Dutch Parallel Reduction Machine Group investigates the feasibility of building a parallel reduction machine suited for the efficient execution of programs written in a functional language. An (abstract) machine which performs reductions is called a *reducer*. The principle of reduction is simple, so there is no doubt that reducers can be built, in software and hardware, sequential and parallel. The *desirability* to do so depends on the advantages of functional languages as a new programming tool, on the possibility to implement them efficiently and to what cost. With these factors in mind, the Dutch PRM-group has concentrated its research mainly on the basic problems of efficient implementation.

First we introduce functional programming languages, discussing advantages, disadvantages and implementation issues. Then we address the important topic of the underlying reduction model. Furthermore we discuss the sequential and parallel implementation of the model we have chosen and the architecture of the experimental parallel reduction machine.

2. Functional Programming Languages

In 1936 two computational models were introduced, one by A. Turing and one by A. Church. Turing described a class of machines (later to be called *Turing* machines). He defined the set of computable functions as those that are implementable on his machines. Based on the concept of a

* This work is sponsored by the Dutch ministry of Science and Education, dienst wetenschapsbeleid

Turing machine are the present day *Von Neumann* computers. *Imperative* programming languages such as FORTRAN, PASCAL etc. as well as all the assembler languages are based on the way a Turing machine is instructed: by a sequence of statements that modify the internal state of the machine.

Church on the other hand invented a formal system called *lambda calculus* and defined the notion of computable function via this system. He did not give indications for implementing his system. *Functional* programming languages, like MIRANDA[1] , HOPE etc., are related to the lambda calculus. An early (although somewhat hybrid) example of such a language is LISP. *Reduction* machines are specifically optimised for the execution of these functional languages.

Also in 1936 Turing proved that both computational models are equally strong in the sense that they define the same class of computable functions.

Rather than giving a precise definition of what lambda calculus or a functional programming language is, we give some examples of functional programs. For a more formal treatment see [BAR86].

A *functional program* consists of an expression E representing both the input and algorithm (this is according to a reduction model similar to the lambda calculus). This expression E is subject to some fixed rewrite rules. *Reduction* consists of replacing a part P of E by another expression P' according to the given rewrite rules. Such an expression P is called a *redex* i.e. a *red*ucible *ex*pression. In schematic notation: E[P] --> E[P'], provided that P --> P' is according to the rules. This process of reduction will be repeated until the resulting expression has no more parts that can be rewritten. This so called *normal form* E* of the expression E will be the output of the given functional program.

Example. $(7 + 4) * (8 + 5 * 3)$ --> $11 * (8 + 5 * 3)$
--> $11 * (8 + 15)$
--> $11 * 23$
--> 253.

In this example the reduction rules consist of the "tables" of addition and of multiplication on the numerals. Note that the "meaning" of an expression is preserved after reduction: 7 + 4 and 11 have the same interpretation. This feature of the evaluation of functional programs is called *referential transparency*. Also symbolic computations can be done by reduction.

Reduction systems for functional languages usually satisfy the *Church-Rosser* property, which implies that the normal form obtained is independent of the order of evaluation of subterms. Indeed, the first example may be reduced also as follows:

$(7 + 4) * (8 + 5 * 3)$ --> $(7 + 4) * (8 + 15)$
--> $11 * (8 + 15)$
--> $11 * 23$
--> 253.

Or even by evaluating several expressions at the same time:

$(7 + 4) * (8 + 5 * 3)$ $\Rightarrow$ $11 * (8 + 15)$
--> $11 * 23$
--> 253.

This gives the possibility of parallel execution.

---

[1] MIRANDA is a trademark of Research Software Limited

2.1. Advantages and Disadvantages of Functional Languages

The most important advantages of functional languages are the following.

- Some algorithms can be described easier in functional languages than in imperative languages. This is caused by the high expressive power of functional languages. For example one may use infinite data structures or yield a function as result of a function.
- The referential transparency makes it possible to use natural mathematical proof methods for verifying a functional program. It also makes program transformations easier: a programmer can start with a straight forward solution of a problem and via transformations convert this solution to a more efficient one. Therefore it is less difficult to develop reliable software.
- Due to the Church-Rosser property, alternative evaluation orders, such as parallel evaluation, will never produce a wrong result, although special care has to be taken in order to avoid non-termination.

However, functional languages also have some disadvantages.

- Functional languages only describe algorithms that transform data into other data. Several applications have interactive aspects and depend on internal states. Such applications like process control, real time programming, data base managers and operating systems can only be implemented on a reduction machine with appropriate interfaces. (On a Von Neumann computer these interfaces are also needed. But for these machines they can be programmed in the same imperative way as the machine itself.)
- Until recently, functional programs ran much slower than equivalent imperative programs and therefore they could not be used as a general programming tool.

In application areas where there is only interaction with the environment but little or no algorithmic transformations, a reduction machine is indeed not better than a conventional machine. However as soon as complex programs are needed, the advantages of functional programming will pay. Practical experience [KOO87a] confirms that functional languages are a good tool to develop reliable software. Current research [BOU85] shows that even in unexpected areas (such as specifying digital and analog circuits) functional languages form a powerful alternative.

2.2. Implementing functional languages on sequential machines

The descriptive power of functional languages is not present in the well-known imperative languages. One of the reasons is that this power is not easy to implement. In particular, problems are caused by:

- the lazy evaluation (call-by-name) which makes among other things infinite data structures possible;
- the possibility to yield a function as result.

Although the use of recursion instead of program loops is elegant, it does not help to increase the execution speed. As a result, a couple of years ago, functional programs ran that slow that no one would program a serious application in a functional programming language, despite of the advantages of these languages. Refining the set of compilation techniques also used for imperative languages, implementers of functional languages generally have learned how to reduce the additional overhead. In general this is done by:

- removing tail recursion where possible, thus transforming function calls to loops;
- using graphs to "share" an argument called by name such that it is evaluated at most once [WAD71, TUR79];

- using stacks where possible (for example with purely arithmetical expressions) instead of creating objects (graph elements) on the heap [JOH84].

Using the techniques described above code for a sequential machine can be generated, of which the efficiency is comparable to the efficiency of code produced for imperative languages.

The following techniques are still being developed:

- efficient heap management (clever garbage collection; destructive update [HUD86b]). Statistical properties of graph reduction concerning heap management are reported in [HAR87b].
- using strictness analysis [MYC81] to detect those arguments in which a function is strict, i.e. predict which arguments of a function can be evaluated before the function is called in order to speed up performance;

Although most of these techniques are well applicable in the general case, it takes a lot of effort to obtain the full gain of them in special cases. Strictness analysis is not fully worked out yet, theoretically and current work does not always include algorithms which are applicable in practice [REN86,87].

New sequential machine architectures have been designed specially geared to functional languages [CLA80, RICH85, KLU79]. As yet, there has been no significant progress in efficiency from this field of research. Until now, by the time a prototype of a special purpose reduction machine was finished, its speed was surpassed by improved implementations on general purpose machines. The NORMA machine design [RICH85] is based on Turner's combinators [TUR79]. The complete design itself has been done using a functional language. However, its performance is poor and show that these combinators do not form the right reduction model.

3. Reduction model.

History has shown that the design of a parallel machine architecture is rather difficult. This holds in particular for a parallel *reduction* machine. One of the problems is caused by the circumstance that some of the optimisations which are needed for an efficient sequential implementation, such as graphs, introduce inefficiencies in a parallel environment. Both theoretical and practical understanding of reduction is necessary to solve this inconsistency. For this reason a reduction model is necessary that contains the essential concepts of functional languages as well as the essential concepts of their implementation. Furthermore we need a language based on this reduction model which we will use as an intermediary between functional languages and reducers. Then it will be possible to reason about differences between languages and their implementations, to prove correctness and to port declarative programs to different (parallel) machines.

We recognize that parallelism is a difficult problem. In our project we have chosen to exploit coarse grain parallelism via the following of steps. The first one is to construct an efficient sequential reducer (see section 3.1 and 3.2). The next one is to use annotations in order to indicate parallelism (see section 4).

We have seen that functional languages have very little in common with the familiar Turing machine model of computation. Some implementers use directly the λ-calculus as the reduction model for these languages [PEY87]. However, if one wants to have the reduction model also close to the implementations, λ-calculus is not a good choice for the following reasons:

- Most implementations are not really based on λ-calculus but on combinatory logic [TUR79, JOH84, COU85].
- As discussed above, graphs are an essential part in any implementation.

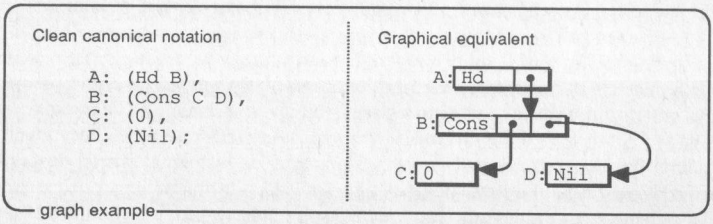
- Patterns in functional languages contain essential information. By translating them to conditionals we might loose some of the information which is essential for the implementation (strictness analysis [NOC87]).
- Functional languages are still being further developed. Several researchers investigate how to incorporate language concepts that have no natural place in λ-calculus such as multi-programming, non-determinism and unification [HUD86a , DEG86].

We have developed two reduction models: Clean [BRU87] and an extension Lean. Clean is an intermediate language between functional languages and their implementations that incorporates all the aspects mentioned. It is an extension of Term Rewriting Systems [O'DO85, KLO85] to Term Graph Rewriting Systems (TGRS). This has the advantage that a lot of theoretical properties from the TRS world are inherited and provide a sound foundation for a TGRS theory. A formal description of the basis and theoretical properties of the reduction model can be found in [BAR87a]. For instance, in [BAR87a] it is proven that all hyper-normalizing reduction (evaluation) strategies in the TRS world, a class to which all well-known normalizing strategies belong, are also normalizing in the TGRS world.

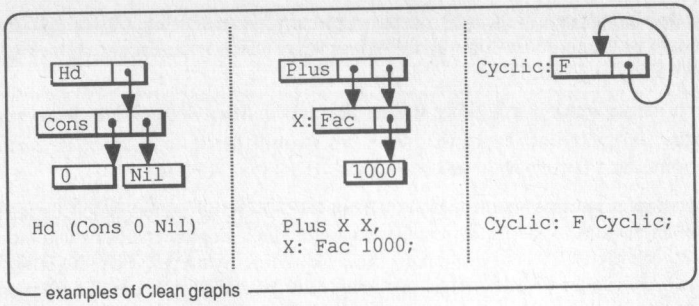
Together with the research group of the University of East-Anglia (UK) which is taking part in the British Flagship project, we are working on a more general language Lean [BAR87b] based on Graph Rewriting Systems (GRS). We hope that this will lead to a more general intermediate language between all kinds of *declarative* languages (such as PROLOG) and (parallel) machines. Preliminary versions of such a language are implemented [GLA87, JAN87]. The research shows that GRS's are a powerful mechanism. We expect that GRS's will be able to incorporate also possible future developments in functional languages such as non-determinism and unification.

3.1. Clean, a functional language based on graph rewriting systems

The object that is manipulated in Clean is a connected, possibly cyclic, directed graph called the *program graph*. Each node in the graph has an unique *identifier* associated with it (corresponding to a machine address) and it contains a *symbol* and a possibly empty sequence of identifiers which define directed arcs to nodes in the graph. Programming with pictures is rather inconvenient so we have chosen for a linear notation for graphs. In the most extensive form of this notation (the canonical form) graphs are represented by giving the list of the nodes out of which the graph is built.



In order to obtain a more readable form we may substitute the contents of a node for a reference to this node and we only explicitly notate the identifiers of nodes if we need them to express sharing. Brackets are left out if they are redundant.



A *Clean program* consists of a set of *rewrite rules* and an initial program graph indicated by Start.Each rewrite rule specifies a possible transformation of the program graph. For instance:

```
Hd (Cons a b)      -> a                        ;

Fac 0              -> 1                        |
Fac n              -> *I n (Fac (-I n 1))      ;

F (F x)            -> x                        ;

Start              -> Hd (Cons (Fac 1000) Nil) ;
```

A Clean graph is rewritten to its normal form by a sequence of applications of individual rewrite rules. For a rule to be included in the sequence, there must be a match between the graph (*redex pattern*) specified at the left-hand-side of the rule and some subgraph (*redex*) of the program graph. A redex pattern matches a redex if both graphs have an isomorphic structure and contain the same constant symbols. If a particular rule is applied to a matching redex, the graph is rewritten according to the right-hand-side of that rule. In general, this right-hand-side also consists of a graph (*contractum pattern*). An instantiation of this pattern (the *contractum*) is constructed and a new program graph is finally constructed by taking all arcs pointing to the root node of the redex and redirecting them to the root node of the contractum. This has the effect of "overwriting" the root of the redex with the root of the contractum.

In general there will be several possible redexes in the graph. It may even be the case that one and the same redex can be reduced according to more than one rule; a situation that is called ambiguity. An algorithm which repeatedly rewrites the graph making choices out of the available redexes and out of all the possible matches of those redexes is called a reduction *strategy*. Given a set of rules (including a start rule), an initial graph and a reduction strategy we have a system with a dynamic behaviour, a reducer. Although it is sometimes only implicitly defined, every implementation of a rewriting system must rewrite according to a given strategy. In principle a Clean program is reduced with a strategy which is called the *functional* strategy, because it resembles very much the way in which reducing is performed normally in lazy functional languages. A formal description can be found in [GOO87]. Alternative strategies may also be used. Via annotations one can give compiler directives to indicate strict arguments or to indicate subgraphs that may be reduced in parallel (jobs for instance, see also section 4.2).

3.2. Using clean to implement functional languages

In [KOO87b] it is shown that functional languages like SASL [TUR79], Miranda [TUR85], OBJ2 [FUT85] and Tale [BAR86] can easily be compiled to Clean code. There are several ways of doing this. For instance, code schemes used by Turner and Hughes [TUR79, HUG82] can be expressed directly in Clean. To demonstrate that compilation is relatively easy we are developing Miranda [TUR85] compilers (one written in Modula2, one written in Miranda) which generate Clean code.

A sequential Clean compiler is developed on a VAX/750 running UNIX BSD 4.2 [BRU87]. We incorporated some of the general implementation techniques mentioned in section 2. Tail recursion is removed, stacks are used where possible. Sharing is controlled by the code generated by the Miranda compiler. Presently we work on two classes of optimisations:

1. those that are understood and are ready to be implemented (improving the use of stacks, pattern matching and heap management);

2. those that require research before they can be implemented (like strictness analysis; we think that Clean is more appropriate to do this than pure λ-calculus [NOC87]).

When the first group of optimisations are implemented we will get a state-of-the-art Clean compiler which will be ported to other sequential architectures.

3.3. Related work

Various Lisp-compilers have reasonable performance, although in general they run not so fast as the current implementations such as Clean. Moreover, Lisp does not support infinite data-structures nor higher order functions.

The Categorical Abstract Machine [COU85] which is based on categorical combinators, is very fast for eager evaluation, but it only supports lazy evaluation as an exception. The same holds for the Hope [BUR80] compiler which uses the special purpose language FP/M [BAI84] as an intermediate language.

The Swedish lazy-ml compiler [JOH84] is on one hand based on a low-level graph manipulation language (the G-machine [KIE85]) and on the other hand it is based on combinatory logic. The performance of the code generated by this compiler is good.

4. Parallel implementation of reduction

Clean has been designed in such a way that it can be fitted by annotation to parallel architectures. This means that the programmer has to mark expressions that can be evaluated in parallel. The annotation may include information about the distribution of the expressions over a network of processing elements.

Parallel implementations of functional languages can be divided into *fine* grain and *coarse* grain. Fine grain parallelism requires only strictness analysis to determine needed redexes, which will constitute the grains. Coarse grain parallelism requires the determination of the grain size of an expression. This size is a measure for the cost of evaluating the expression. Several approximations of the grain size of an expression have been proposed [HUD85, HUG82, KEL79]. However, this function is non computable and to our knowledge there exists no satisfactory heuristic. Because the hardware that is necessary for a fine grain parallel machine is very complex [MAG79], we have chosen for a coarse grain approach. The programmer has to design process- and data-structures in such a way that they can be annotated to match the coarse grain architecture [VRE87].The grain annotation is covered in the next paragraphs.

Clean is being implemented on the Distributed Object Oriented Machine [ODIJ85] developed at the Philips Laboratories, the Netherlands, and on our Experimental Parallel Reduction Machine.

4.1 Local memory architecture

The basic model of the kind of parallel machine architecture that we will investigate is a collection of reduction processors, each equipped with its own local memory, interconnected by a (fast) data communication network. A fundamental property of this model is that the access time of a processor

to its own memory is much shorter than the access time to the other memories. These two access times may differ by one to two orders of magnitude. To account for this property we have decided not to support a global address space i.e. when a subgraph happens to reside in a particular local memory, the nodes of this subgraph can only point to nodes within the same local memory. A separate mechanism is provided to transport a subgraph from one local memory to another.

An experimental parallel reduction machine has been constructed, based on a local memory architecture [HAR86]. The main objective of this effort is to collect statistical data of the communication cost of job based subgraphs (see section 4.2). These measurements can only be efficiently performed on dedicated hardware. The reduction processors have been realized with commercial shared-bus microprocessor systems (M68000 with VME-bus). Communication of graphs has been implemented with the aid of fast parallel bus-interconnections between dual-ported memories (VME to VMX). To exploit the full advantage of these fast connections, the construction of special graph collection processors is envisaged. At present, experiments are being conducted (graph collection in software) to obtain preliminary performance figures of the proposed parallel reduction model on the experimental machine. The combination of fast sequential reduction (Clean) and fast hardware supported communication of coarse grain jobs will in our opinion enable efficient parallel reduction.

4.2. Job based parallel reduction

The architectural features discussed above limit the kind of expressions that can be efficiently reduced in parallel. Only certain coarse grain parts of the program that we call "jobs" are allowed to be copied to another processor for parallel evaluation. A job is an expression with the following properties:

a) it is *self-contained* (i.e. a subgraph containing no references to other points of the graph);

b) its evaluation is needed to compute the final result;

c) the cost to evaluate the expression outweighs the cost involved in transportation.

Job property c) guarantees that parallel execution of a set of jobs will be faster than their sequential execution. Property b) makes sure that the result of a job is essentially used in the whole computation and so no actual processing will be wasted. Finally property a) allows jobs to be evaluated in a separate address space and avoids the need for global garbage collection.

The restriction of parallel reduction to jobs defines a minimum granularity on which the data communication of the architecture can be based. Small quantities of data (like a single node or a redex containing only pointers to its arguments [WAT84]) will never be transported. Overhead incurred by transmission protocols can be spread over the cost of transporting a whole subgraph.

A disadvantage of job based parallel reduction on a local memory architecture is that sharing of expressions cannot be exploited globally across jobs. Within a job all sharing can be maintained. Summarizing, parallel job reduction employs copying reduction on the global job level and sharing reduction within jobs.

4.3. Sandwich reduction strategy

To avoid the major disadvantage of copying (duplication of work), a special reduction strategy has been devised on the job level. This strategy guarantees that a job will be a primary redex when it is transported to another processor. Thus, before copying takes place, a job contains no secondary redexes and hence no work can be duplicated while shipping the job.

In practice the sandwich strategy has been implemented by a single special annotation given by the programmer:

sandwich G job_1 job_2 . . . job_n

where $job_i = F_i\ a_1\ a_2 \ldots a_m$.

The sandwich construct is the only means in the language to create jobs. An expression is sequentially reduced to normal form until a sandwich expression is needed. The reduction of (G job_1 job_2 ... job_n) is then suspended until the parallel evaluations of job_1 job_2 . . . job_n have been completed. However, before the jobs are submitted for parallel evaluation, all arguments a_1 a_2 . . . a_m of each job_i are sequentially reduced to normal form, rendering job_i a primary redex. Now copying the normal forms of the a_i, in order to ship the job, cannot result in extra work. If a_1 a_2 . . . a_m would have been reduced in parallel, then any redex shared between the a_i would have been copied. This would result in the duplication of work. The strategy has been called "sandwich strategy" because it contains one level of eager evaluation between two levels of lazy evaluation.

We have constructed a simulation of a parallel reducer [LEE85,HAR87c]. A number of divide-and-conquer algorithms, such as the fast Fourier transform, have been programmed to make effective use of the sandwich strategy.

If the volume of data processed by an algorithm is characterised by a number "n" then the algorithms that have a computational complexity > O(n), can be made to benefit significantly from parallel evaluation with the sandwich strategy.

The speed-up that may be achieved for a particular value of "n" depends on some architectural properties, in particular the ratio between the sequential execution speed of a processing element and the transfer speed of jobs and results. For a sufficiently large value of "n", a near linear speed-up results as more processing elements are used. In this case, the speed-up is only dependent on the number of processing elements.

For the algorithms that have a computational complexity $\leq O(n)$, the speed-up that may be achieved is also dependent on the ratio defined above.

The speed-up for divide-and-conquer algorithms is also valid for other architectures with local memory [MCB87, KLU83].

Experiments on our experimental parallel reduction machine have been conducted in order to measure the architecture constant and to study its effect on divide-and-conquer algorithms [HAR87a].

4.4. Related work

Most proposals for parallel reduction are either based on pure copying reduction [KLU83, MAG79, TRE82], or on pure sharing reduction [DAR81, HUD85, KEL79, PEY86, WAT84]. To our knowledge mixed reduction based on both copying and sharing has not been proposed.

Two contemporary and similar research projects are the Flagship machine [WAT84] and the Grip project [PEY86]. In contrast to our proposal both machines feature a global address space. Like the Alice machine [DAR81] the Flagship machine is a packet rewrite machine and will have rather fine grain communications. The Grip machine is based on super combinator reduction and will use conservative parallel strategies. To compensate for the lack of a global address space we spent much effort in the optimisation of job based graph-communication. These projects are in a phase too early to make comparison of results possible.

5. Future research

In cooperation with the research group of the University of East-Anglia the reduction model will be investigated further in order to include modularization, general type system, unification, general I/O etc. Also we will investigate the possibility for the user to define his own special (parallel) strategy for his program, for instance via high level specification of (parallel) reduction strategies and a formalism for mixing several strategy schemes during evaluation [EEK86]. This research must make the language Lean [BAR87b] based on this GRS a good intermediate language between declarative languages and (parallel) machine architectures. All this must be accomplished without loosing the basic elegance, the practical usability and the theoretical framework of the model. In the near future we will improve the practical applicability of Clean and Lean by increasing the efficiency and by adding separate compilation, typing and strictness analysis.

The efficiency of the experimental parallel reduction machine can be increased by the construction of special hardware components for specific tasks such as graph compaction and packet transport and we will try to extend the class of problems for which the sandwich strategy is applicable. Ideas concerning load distribution are presently being tested by experiments on the experimental parallel machine.

6. Conclusions

- We have designed a reduction model, a Graph Rewriting System (GRS) which includes the most relevant aspects of functional languages and the way they are implemented. With this model we can derive *theoretical* properties which are valid for parallel reducers.
- We have developed the experimental language Clean based on this GRS. It is used as an intermediate language between functional languages and (parallel) machine architectures. With Clean *practical* properties of reducers can be examined and compared. Functional languages can be translated in a natural way into Clean.
- The efficiency of the Clean implementation is good (comparable to PASCAL).
- For divide-and-conquer algorithms it is possible to obtain significant increase of performance on a parallel reduction machine. Reduction is not essential for this speed-up, but it makes parallel programming much easier.
- The sandwich annotation is a good way to implement parallel reduction on a local memory architecture.

Finally we give a status quo answer to the question stated in the abstract :

It is possible and realistic to construct an efficient sequential reduction machine on existing sequential hardware. For a particular class of algorithms it is possible and realistic to implement a parallel reduction machine on existing parallel architectures (direct- and shared-connection machines). Further speed-up can be achieved by introducing special hardware components. This may eventually lead to the construction of a special purpose reduction machine.

7. Acknowledgements

The following people of the Dutch parallel reduction machine project have made valuable contributions to the work presented in this paper: Jan Bonneveld, Raymond Boute, Tom Brus, Maarten van Leer, Marc van Leeuwen, Hans Mulder, Betsy Pepels, Gerard Renardel de Lavalette, Arthur Veen and Zhao Jing-Wei. We are grateful for the collaboration with the University of East-Anglia.

8. References.

[BAI84] Bailey, R., "FP/M Abstract Syntax Description", Internal Document, Imperial College London, 1984.

[BAR86] Barendregt, H.P., Leeuwen, M. van, "Functional Programming and the Language Tale", (Eds. J.W. de Bakker, W.-P. de Roever and G. Rozenberg), Springer LNCS 224, pp. 122 - 207, 1986.

[BAR87a] Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R., "Term Graph Reduction", University of East-Anglia and University of Nijmegen, Proceedings of Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, the Netherlands, LNCS vol 259, pp. 141-158, June 1987.

[BAR87b] Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R., "Towards an Intermediate Language based on Graph Rewriting", University of East-Anglia and University of Nijmegen, Proceedings of Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, the Netherlands, LNCS vol 259, pp. 159-175, June 1987.

[BOU85] Boute, R.T., "Functional description of digital systems", in: IFIP WG 10.1 Working conference on Methodologies for Computer System Design, North-Holland, Amsterdam, 451-467.

[BRU87] Brus, T.H., Eekelen, M.C.J.D. van, Leer, M.O. van, Plasmeijer, M.J., "Clean, A Language for Functional Graph Rewriting", University of Nijmegen, Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87), Portland , Oregon, USA, LNCS 274, pp. 364-384, September 1987.

[BUR80] Burstall, R.M., MacQueen, D.B., and Sanella, D.T., "Hope: An Experimental Applicative Language", Proceedings of the 1980 LISP Conference, pp. 136-143, Stanford, California, August 1980.

[CLA80] Clarke, T.J.W., Gladstone, P.J.S., MacLean, C.D., and Norman, A.C., "SKIM - the S,K,I Reduction Machine", Proceedings of the 1980 ACM Lisp Conference, pp. 128-135, August 1980.

[COU85] Cousineau, G., Curien, P.L., Mauny, M., "The Categorical Abstract Machine", Proc. Conf. on Functional Languages and Computer Architecture, Nancy, pp. 50 - 64, September 1985.

[DAR81] Darlington, J., Reeve, M., "ALICE: A multiple-processor reduction machine for the parallel evaluation of applicative languages", ACM Conference on functional programming, languages and computer architecture, pp 65-76, New Hampshire, Oct. 1981.

[DEG86] De Groot, D. & Lindstrom G. (eds), "Logic Programming: Functions, Relations and Equations", Prentice Hall 1986.

[EEK86] Eekelen, M.C.J.D. van, Plasmeijer, M.J., "Specification of rewriting strategies in Term Rewriting Systems", University of Nijmegen, Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico, LNCS vol. 279, October 1986.

[FUT85] Futatsugi, K., Goguen, J., Jouannaud, J-P. and Meseguer, J., "Principles of OBJ2", Proceedings of 12th ACM Symposium on Principles of Programming Languages (POPL '85), pp. 52-66, ACM, 1985.

[GLA87] Glauert, J.R.W., Kennaway, J.R., Sleep, M.R., "DACTL: A Computational Model and Compiler Target Language Based on Graph Reduction", ICL Technical Journal, May 1987.

[GOO87] Goos, J., Van Latum, F., "The use of an abstract-interpretation TRS in specifying and proving reduction strategies", Master Thesis, University of Nijmegen, March 1987.

[HAR86] Hartel, P.H., Vree, W.G., "A Load Distribution Network for a Multi Processor Reduction Machine", Internal Report D-6, Dutch Parallel Reduction Machine project, University of Amsterdam, April 1986.

[HAR87a] Hartel, P.H., Vree, W.G., "Parallel graph reduction for divide-and-conquer applications", Internal Report D-15, Dutch Parallel Reduction machine project, University of Amsterdam, Feb 1987.

[HAR87b] Hartel, P.H., Veen, A.H., "Statistics on graph reduction of SASL programs, March 1987, to appear in Software Practice and Experience.

[HAR87c] Hartel, P.H., "Experimental Parallel Reduction Machine User Manual", Internal Report D-14, Dutch Parallel Reduction Machine Project, University of Amsterdam, Oct 1987.

[HUD85] Hudak, P., Goldberg, B., "Distributed execution of functional programs using Serial combinators", IEEE Transactions on computers, Vol. C-34, 10, pp 881-891, Oct. 1985.

[HUD86a] Hudak, P., and Smith, L., "Para-Functional Programming: A Paradigm for Programming Multi-Processor Systems", 12th A.C.M. Symp. on Principles of Programming Languages, pp. 243 - 254, Jan. 1986.

[HUD86b] Hudak, P., "Arrays, Non-Determinism, Side-effects and Parallelism: A Functional Perspective", proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico, LNCS vol. 279, October 1986.

[HUG82] Hughes, J., "Graph reduction with super-combinators", rep PRG-28, Oxford University, June 1982.

[JAN87] Jansen, T., "Interpreting Lean", M. Sc. thesis, University of Nijmegen, May 1987.

[JOH84] Johnsson, T., "Efficient Compilation of Lazy Evaluation", Proc. of the ACM Sigplan '84, Sigplan Notices, Vol. 19, No 6, June 1984.

[KEL79] Keller, R.M., Lindstrom, G., Patil, S., "A loosely-coupled applicative multi-processing system", Proc. Nat. Comp. Con., Vol. 48, pp 613-622, 1979.

[KIE85] Kieburtz, R.B., "The G-machine: A fast, graph-reduction evaluator", Proc. Conf. on Functional Languages and Computer Architecture, Nancy, pp. 1 - 16, September 1985

[KLO85] Klop, J.W., "Term rewriting systems", to appear in the Handbook of Logic in Computer Science, editors S. Abramsky, D.M. Gabbai, T.S.E. Maibaum.

[KLU79] Kluge, W.E., "The Architecture of a Reduction Language Machine Hardware Model", GMD, Schloß Birlinghoven, D-5205 St. Augustin 1, Interner Bericht ISF-79-3, 1979.

[KLU83] Kluge, W.E., "Cooperating reduction machines", IEEE Transactions on computers, Vol. C-32, 11, pp 1002-1012, Nov. 1983.

[KOO87a] Koopman, P.W.M., "Interactive Programs in a Functional Language: A Functional Implementation of an Editor", Software Practice and Experience,Vol 17(9), pp 609-622.

[KOO87b] Koopman, P.W.M., Nocker, E.G.J.M.H., "Compiling Functional Languages to Functional Graph Rewriting Systems", University of Nijmegen, to appear.

[MAG79] Mago, G.A., "A network of microprocessors to execute reduction languages", Part I and II. International journal of computer and information sciences, Vol. 8, 5, pp 349-385 and 6, pp 435-471. 1979.

[LEE85] Lee, Th.G.A. van der, Oolman, H., "Tracing and debugging in SASL", Report 66, Department of Informatics, Nijmegen University, April 1985

[MCB87] McBurney, D.L. and Sleep, M.R., "Transputer-based experiments with the ZAPP architecture", Proceedings of Parallel Architectures and Languages Europe (PARLE), part I, Eindhoven, the Netherlands, LNCS vol 258, pp. 242-259, June 1987.

[MYC81] Mycroft, A., "Abstract interpretation and optimising transformations for applicative programs", PhD. thesis, Univ. of Edinburgh, 1981.

[NOC87] Nocker, E.G.J.M.H., "Strictness Analysis for Functional Graph Rewriting Systems", University of Nijmegen, to appear.

[O'DO85] O'Donnell, M.J., "Equational Logic as a Programming Language", Foundations of Computing Series, MIT Press, 1985.

[ODIJ85] Odijk, E.A.M., "DOOM: a Decentralized Object-Oriented Machine", Doc. Nr. 0125, Esprit 415 internal report, Philips, Eindhoven, 1985.

[PEY86] Peyton Jones, S.L., "Using future bus in a fifth generation computer", Microprocessor and microsystems, Vol. 10, 2, pp 69-76, March 1986.

[PEY87] Peyton Jones, S. L., "FLIC - a Functional Language Intermediate Code", Dept. of Comp. Sc., University College London, internal working paper.

[REN86] Renardel de Lavalette, G., "Strictheidsanalyse", Department of Philosophy, University of Utrecht, Logic Group Preprint Series, no. 10, May 1986.

[REN87] Renardel de Lavalette, G., "Strictness analysis for a language with polymorphic and recursive types", University of Utrecht, to appear.

[RICH85] Richards, H., "An Overview of the Burroughs NORMA", Proceedings of the Workshop on Implementation of Functional Languages, Programming Methodology Group University of Göteborg and Chalmers University of Technology, Report 17, pp. 429-438, February 1985.

[TRE82] Treleaven, P.M., Hopkins, R., "A recursive computer for VLSI", Proceedings of the 9-th international IEEE/ACM symposium on computer architecture, special issue of Computer architecture news, pp 229-238, Austin, Texas, 1982.

[TUR79] Turner, D.A., "A new Implementation Technique for Applicative Languages", Softw.

Pract. and Experience, Vol 9 (1), pp. 31 - 49, January 1979.

[TUR85] Turner, D.A., "Miranda: A non-strict functional language with polymorphic types", Proc. Conf. on Functional Languages and Computer Architecture, Nancy, pp. 1 - 16, September 1985.

[VRE87] Vree, W.G., "The grain size of parallel computations in a functional program", Proc. Int. Conf. on Parallel Processing and Applications, L'Aquila, Italy, September 1987.

[WAD71] Wadsworth, C.P., "Semantics and Pragmatics of the Lambda-calculus", Ph. D. thesis, Oxford University, 1971.

[WAT84] Watson, I., "A packet based data-driven rewrite-rule machine", Department of computer science, University of Manchester.

A RECURSIVELY SCALABLE NETWORK VLSI IMPLEMENTATION

G. Della Vecchia and C. Sanges

Hybrid Computing Research Center - C.N.R.
via Claudio 21, 80125 Napoli - ITALY
phone n. +39-(0)81-636683

Object of the present work is to briefly introduce a new general class of recursively scalable network topologies for message-passing architectures, referred to as WK-recursive topologies, proposed by the Authors, and to describe the VLSI implementation of a regular, direct WK-recursive based network realized at the Hybrid Computing Research Center.

1. INTRODUCTION

During the last years evidence has been shown that none of the machine organizations of commercially available supercomputers can fairly cover the whole spectrum of possible applications. An effort to extend the range of applications should be oriented toward a composite organization relying on several types of concurrency and granularity. It should offer high parallelism and scalability, exhaustively exploiting VLSI and communication technologies and should efficiently and reliably support very high level programming languages (logic, functional and object oriented) as well as high performances in scientific numeric computations and symbolic applications. In this context massively parallel multi-style architectures seem to represent the key factor for future generation computers [1]. Based on this approach, the CAPRI Project (Concurrent Architecture and PRogramming environment for highly Integrated system) is aimed to the definition and the implementation of a multi-style massively parallel computer. Supported by the Strategic Program on Parallel Computing of the National Research Council of Italy, the CAPRI

Project is under development at the Hybrid Computing Research Center - CNR, Naples (Italy) in cooperation with the Department of Computer Science - University of Pisa, Pisa (Italy) [2].

A first phase of the project consists in the realization of a prototype [3] which is intended to be a massively parallel system based on the concurrent computational model of communicating processes having message-passing as data mechanism and control-driven as control mechanism [4]. It has been designed to support in an efficient manner an high level concurrent language, the ECSP (Extended CSP) [5], and relies on a microcomputer array implementation whose VLSI components belong to INMOS transputer's family. The design of such a distributed system involving many processors which cooperate according to the message-passing computational model imposed the choice of a suitable communication network topology which, ultimately, would be for a great amount responsible of the system overall performance. As widely confirmed in literature [6], a valid interconnection networks has to meet some requirements of regularity, locality and expansibility, at least as far as message-passing approach is concerned. There, nodes produce, consume and route messages so that they form what are called direct networks which constitute the core of experimental VLSI multi-microcomputer systems, included the one this study deals with.

On behalf of the design of the prototype, a new general class of recursively scalable network topologies for message-passing architectures, referred to as WK-recursive topologies, has been proposed and detailed described by the Authors in a previous work [7].

Object of this paper is to present the VLSI implementation of a regular, direct network derived from a topology belonging to the WK-recursive class and carried out at the Hybrid Computing Research Center.

In the course of the present work, the next paragraph is devoted to a brief description of the WK-recursive general class of topologies and their basic peculiarities, while in paragraph 3 the actual implemented network is discussed. Then, the internal architecture of the communication node organized as array of cooperating processes is sketched in paragraph 4 where the message exchange algorithm based on a self-routing technique is also shown. Paragraph 5 illustrates the hardware/software environment adopted for implementing the network, together with a

rough description of the VLSI basic component used, the T414 INMOS transputer, and the OCCAM concurrent programming language it supports. Furthermore, paragraph 6 contains the OCCAM description of the network and the OCCAM processes which implement the communication node as well as the technique used to map OCCAM channels through which processes exchange information onto transputer's physical links. Finally, concluding remarks are given in paragraph 7 where developments in the realization of the prototype planned for the immediate future are mentioned, too.

2. WK-RECURSIVE TOPOLOGIES

In the previous mentioned work, we defined a class of regular, scalable topologies organized according to virtual nodes pertaining to different expansion levels obtained as recursive replications of a basic fully connected structure (i.e. the first-level virtual node). Virtual nodes are built in a manner similar to the application of "compounding operations" as reported in [8], on the basis that as the number of levels varies the topological characteristics remain unchanged. Because of the recursive nature of these topologies whose peculiar aspect is the equality between the amplitude W and the degree k of virtual nodes we decided to designate such a class "WK-recursive".
A member belonging to the class of WK-recursive topologies is identified essentially by three parameters :

N = number of real nodes
k = node degree
L = expansion level

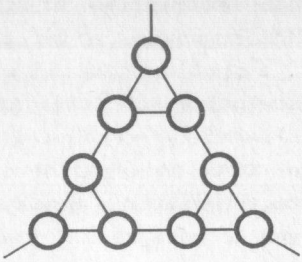
for which holds the following analytical relation :

$$L = \log_k N \qquad (1)$$

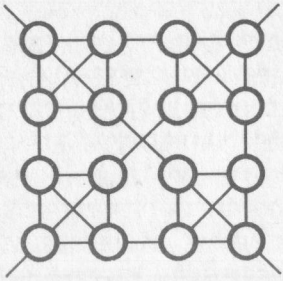
This relation allows the definition of topology characteristic indices in the case of WK-recursive class. For instance, the diameter is given by the formula

$$D = 2^L - 1 \qquad (2)$$

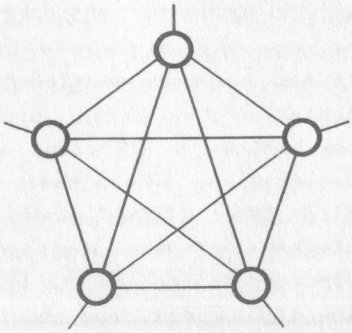
k = 2 L = 2 N = 4



k = 3 L = 2 N = 9



k = 4 L = 2 N = 16



k = 5 L = 1 N = 5

Fig. 1

Clearly, this quantity does only depend on the expansion level whatever the node degree is, and this applies to all WK-recursive topologies, some examples of which are illustrated in fig.1 where it is worth noting the special cases of k=2 and k=N which lead to the linear array and a fully connected configuration, respectively.

Main features characterizing this class of topologies can be summarized in the following focal points:

- WK-recursive based networks offer an high degree of regularity and scalability which very well conform to a modular design and realization of distributed systems involving large number of computing elements, since they give true scalability "at hands" in terms of hardware implementation

- a network of any arbitrarily large size can be built using nodes with any specified number of links per node
- internode distances are kept very small
- networks admit self-routing techniques for message exchanges based on an expansibility without reprogramming approach
- networks show an high degree of local density which thus allows subnetwork clusters to be outlined: this can suggest the utilization of suitable strategies for balancing the computational load.

Moreover, they provide enough flexibility to make an optimum choice of the topological parameters which often has to meet contrasting requirements (e.g., low node degree and small internode distance [9]). Finally, due to their organization according to virtual nodes, it is possible to devise plain techniques for special process-to-processor allocation along with broadcasting and error confinement to which local environment, message-passing approaches are very sensitive.

3. THE IMPLEMENTED NETWORK

In a direct network each node, which can be implemented by one or more processors, is an active element provided with the required amount of computational resources able to manage the communications, the distributed operating system local instances and the applications. This model has been adopted in the design of the prototype, where the network, of homogeneous type for the sake of the modularity, has a WK-recursive topology with a node degree k=4, and each node is realized by several interconnected VLSI processors committed to distinct level of functionality: communication, kernel run-time support for the ECSP environment and user application.
The realization described in the present paper was intended to carry out the communication level of functionality, which constitutes the infrastructure of the whole parallel system, as well as the implementation of a message exchange technique based on a self-routing algorithm purposely devised for WK-recursive topologies. For this reason we shall deal with a network of communication nodes each of them being actually implemented by a single physical processor.

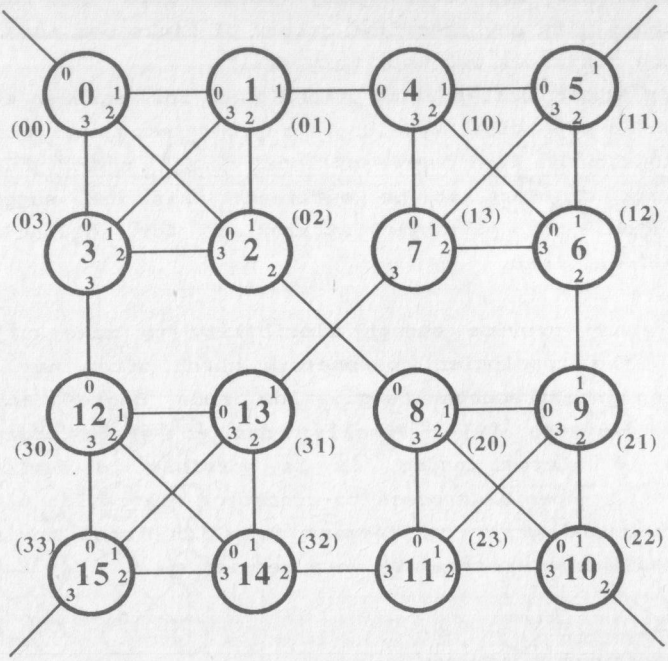


Fig. 2

The network extension reaches the second level of expansion, for a total of 16 nodes (cfr. (1)) and is schematized in fig.2. The node indexing scheme, as described in [7], has been established as follows. Once an origin and an orientation have been arbitrarily fixed, each (l-1)-th level virtual node composing a l-th level virtual node is assigned an index n_l. As a result, in a network expanded at level L composed of N real nodes each of them is characterized by a L-plet of indices $(n_L,\ldots,n_2,n_1)$. A more compact notation for real nodes can be achieved by considering each index as being assigned a weight according to the expression

$$n = \sum_{l=1}^{L} n_l \, W^{l-1} \qquad \text{resulting } n \in \{0,\ldots,N-1\}$$

therefore each node can be univocally identified by a node number n which can be regarded as the decimal coding of the W-ary number $n_W = n_L \ldots n_2 n_1$. In this way, for the given network a couple of indices shall suffice for numbering all the nodes.

As far as the communication link indexing is concerned, it is local to each of the W real nodes constituting the first level virtual node and is based on their indices n_1. One real node has k links through which communications take place. Since k=W, k-1 of them are used to connect the node with the remaining W-1 nodes in the first level and one is left free; each of the k-1 links is numbered according to the value of the index n_1 of the node it is connected to, and the one left free is given the number equal to the value of the index n_1 of the node it belongs to.

4. COMMUNICATION NODE ARCHITECTURE

As introduced above, each of the sixteen nodes of the network prototype has been implemented by means of a VLSI processor entirely dedicated to communications. Two external channels (i.e., channels established between processes allocated on different processor) are mapped onto each of the four bidirectional communication links, one input channel and one output channel. Every input channel is handled by an input process as well as every output channel is handled by an output process, so that eight processes, residing in the node processor and communicating each other through internal channels as shown in fig.3, will run concurrently. To be rigorous, since this eight processes stay on the same processor, they will run actually in a quasi-concurrent fashion, under the control of the processor built-in scheduler. Yet, due to the particular architecture of the transputer (cfr. next paragraph) and the relatively limited amount of resources required by the I/O processes it is not jeopardous to consider them running in effective parallelism.
This communication node architecture seemed to deliver the best chance to fully exploit the intrinsic parallelism offered by the transputer which, incidentally, is equipped with two DMA channels per link. The output process waits for the occurrence of a message coming from any of its internal channels and then forward it to the external output channel. The input process waits for a message to come along its external input channel and, on the basis of the information contained in the message itself (i.e., the destination node index) routes it to the proper output process through an internal channel. The routing decision is based only on the content of the message which is never altered

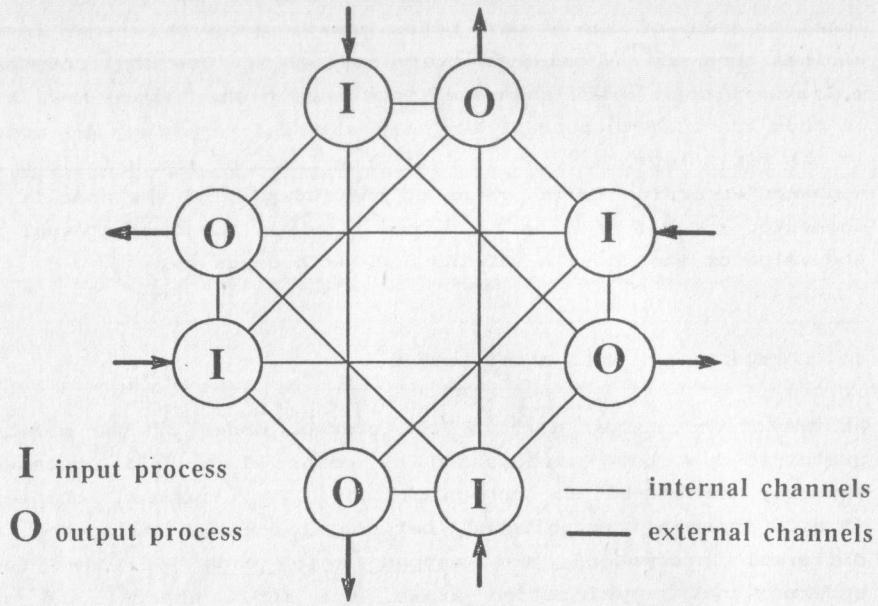


Fig. 3

along the path; this technique is often referred to as "self-routing" in literature [10]. A message is formatted at the source node with the destination node index embedded into, where the destination node d is specified in the W-ary notation $d_W = d_L \ldots d_2 d_1$; hence if the message arrives at the current node t the routing takes place as follows:

- compare t_W to d_W
- if they match then the message reached its destination else
- forward the message to the output process which handles that output channel mapped on the link number equal to the most significant digit of d_W different from t_W.

This algorithm, which is effective for the whole class of WK-recursive topologies, has been devised in order to route messages along the shortest path connecting source and destination nodes so that at most D (cfr. (2)) routing steps are required.

5. HARDWARE-SOFTWARE ENVIRONMENT

Current VLSI technology offers single chip microcomputers with built-in memory and communication links which quite obviously constitute eligible basic components for direct network implementations. In particular, the INMOS Ltd. had presented a family of programmable VLSI components, the transputer [11], whose features sounded satisfactorily fitting the requirements for implementing a WK-recursive based network which, at the uttermost, will have to count a high number of nodes. A typical member of transputer's family is a single chip containing processor, memory and communication links which provide point to point connection between transputers. Its architecture has been designed to permit a network of programmable components to have any desired topology, limited only by the number of links on each transputer minimizing the constraints on the size of such a system. Due to the presence of point to point communication links, there is no contention for the communication mechanism, regardless of the number of processors in the network and, this aspect being significantly important, the communications bandwidth does not saturate as the size of the system increases. Rather, the larger the number of processors in the system, the higher the total communications bandwidth of the network.

We used the IMS T414, a member of transputer's family, equipped with 2K static RAM and four bidirectional 10 Mbit/s serial links connected to eight DMA channels handled concurrently. It provides a direct implementation of the process model of computing supporting a set of concurrent processes, each with own program and data, time-shared by a built-in scheduler, and communicating with other simultaneous processes using explicitly defined channels. The number of registers which hold the process context is small and this, combined with fast on-chip RAM, provides a sub-microsecond process switch time.

Like any transputer, the IMS T414 supports the Occam programming language [12] which allows applications to be described as a collection of processes operating concurrently, in a fashion very similar to Hoare's Communicating Sequential Processes programming language [13], even though at a lower abstraction level. Processes communicate by means of channels: each Occam channel provides a one-way communication path between two concurrent processes. Communication is synchronized and takes place when

both the inputting process and the outputting one are ready. A channel between two processes executing on the same transputer (internal channel) is implemented by a single word in memory whereas a channel between processes running on different transputers (external channel) is implemented by point to point links. A program running in a transputer is formally equivalent to an Occam process, so that a network of transputers can be described directly as an Occam program.
As a basic component of development systems, the transputer requires the support of a general purpose host computer. For this motive, we have been using a Digital's VAX 11/750 for which a full Occam/Transputer development environment has been made available by Inmos. The Occam Programming System (OPS) [14] simulates the network behaviour by implementing it on the host computer mapping the Occam description onto single- or multiple-processor target hardware, analyzing failures and producing documentation, all on an interactive basis. On the other hand, the Transputer Development System (TDS) [15] provides facilities for loading and monitoring a transputer network.

6. OCCAM IMPLEMENTATION

The communication node constitutes the active entity of the direct interconnection network, providing a mechanism for the message exchange activity occurring among the higher level processes running on the transputers constituting the parallel system. In our implementation it is described in Occam by a process "node" which reflects the architecture illustrated in paragraph 4. All the network external channels are gathered in an array: each copy of the process "node" sees a unique eight-element slice of such an array (four input and four output external channels) whose actual index values are passed to it at system initialization and are a function of the index of the node it is allocated to. The process "node" includes the definition of the array of input and output processes together with the declaration of the internal channels they employ, and their parallel activation. In addition, a "local process" has been appended for the purpose of emulating the presence of a process handling the communications toward the processing nodes which in the prototype will host processes of the kernel run-time support

for the specific ECSP application. Actually, this "local process" acts as a consumer process which accepts messages addressed to the node and, for testing purposes, sends them back to their respective senders having affixed a mark which in practice is the node index.

The network is configured by allocating sixteen copies of process "node" to as many processors and mapping the external channels onto processors links. The process to processor allocation is specified by the Occam constructs PLACED PAR and PROCESSOR which tell the Occam kernel which process is to be allocated to what processor. In particular, the use of the replicator construct

```
PLACED PAR n = [0 FOR number.of.processors]
```

allows the allocation of a copy of process "node" to the processor being assigned the index n.

The mapping of external channels onto the processor links is specified by the construct PLACE: each channel must be placed twice, at an input link address on one processor and at an output link address on a different processor. That is the way the network is described. Index values pertaining to input channels are computed in the following way:

$$\text{inp.ind} = \text{link.number} * K^G + n_G * \sum_{l=0}^{G-1} K^l + \sum_{l=G+1}^{L} n_l * K^l$$

where G stands for the link degree, which we defined as the expansion level number of the virtual node the link belong to; in other words, a link connecting two virtual nodes of level L-1 has a degree of L. Likewise, index values pertaining to output channels are computed in the following way:

$$\text{out.ind} = \text{link.number} + \sum_{l=1}^{L} n_l * K^l$$

These expressions permit the evaluation of the channel array indices whatever the network expansion level is. This important facility means that once the first level virtual node has been specified, a further network expansion to an higher level does not require any change in the Occam description, being only necessary to update the constant value "number.of.processors" at Occam PROGRAM level. Below are listed the Occam source code of

the network configuration, the process "node" and the input and the output processes.
After the network implementation was completed, it was necessary to arrange things in such a way as to monitor the network behaviour and test the effectiveness of the routing algorithm. For this aim, we used an additional IMS T414 transputer acting as a front-end processor with its four links connected to those left free in the network. In addition, we attached a CRT terminal to an RS-232 serial port directly interfaced to the transputer's bus. This front-end processor hosts Occam software to manage the CRT terminal and a process "front.end" which produces and sends messages to one or more network nodes as interactively specified through the terminal. To trace the paths followed by the messages, the input process has been slightly modified so that the content of these latter is updated with the index of the nodes they pass through.
In this way, a process "tracer" residing on the front-end processor shows tracing information on the terminal which, being equipped with color graphics facilities, allows a network representation to be displayed on the screen in order to highlight the path followed by the messages.

7. CONCLUSIONS

This work describes the VLSI implementation of the communication functionality constituting the framework of a prototype which is meant to be a first step toward the realization of a massively parallel multi-style system, an attempt to overcome problems arising from the narrow range of applications which is typical of current parallel systems. The communication medium here discussed relies on a direct, homogeneous interconnection network whose topology belongs to the WK-recursive general class of topologies early suggested by the Authors. WK-recursive networks, named after their particularity of having the node degree equal to the amplitude of virtual nodes they are organized into, offer several properties which profitably meet the requirements postulated in the design of a massively parallel system. The first WK-recursive prototype concerns a network expanded at the second level, for a total of sixteen nodes; a further expansion will be easily achieved since the Occam code describing the network and the

communication processes need not to be re-written, being applicable to whatever topology belonging to the WK-recursive class. Currently, we are testing a broadcasting technique for the WK-recursive topologies, described in [16], which requires a number of steps equal to the network diameter. Our further efforts will be oriented toward the definition and the implementation of fault tolerant routing and broadcasting algorithms.

REFERENCES

[1] F. Baiardi, M. Vanneschi, "Parallelism Issues in Multi-Style Computers", ESPRIT Summer School on Future Parallel Computers, June 9-20, 1986, Pisa, Italy

[2] R. Vaccaro, M. Vanneschi et al., "An Integrated Approach to the Design of a Massive Parallelism System", Proceedings of The International Conference on Parallel Processing and Applications, Sept. 23- 25, 1987, L'Aquila, Italy

[3] G. Della Vecchia, C. Sanges et al., "HCRC-parallel computer Prototype: Architecture and Programming Techniques", Proc. 11th IMACS World Congress on System Simulation and Scientific Computation, Aug. 5-9, 1985, Oslo, Norway, vol.5, pp. 385-388

[4] P.C. Treleaven, I.G. Lima, "Future Computers: Logic, Data Flow,...,Control Flow?", Computer, March 1984, pp. 47-55

[5] F. Baiardi, A. Fantechi, M. Vanneschi, "Language Constructs for a Robust Distributed Environment and their Semantics", The Muteam Experience in Designing Distributed Systems of Microprocessors, CNR, Dec.1983

[6] C.L. Seitz, "Concurrent VLSI Architectures", IEEE Trans. Computers, Dec. 1984, Vol. C-33, No.12, pp.1247-1265

[7] G. Della Vecchia, C. Sanges, "Recursively Scalable Networks for Message Passing Architectures", Proceedings of The International Conference on Parallel Processing and Applications, Sept. 23-25, 1987, L'Aquila, Italy

[8] L. Uhr, "Algorithm-Structured Computer Arrays and Networks", Academic Press Inc., 1984

[9] L.N. Bhuyan, D.P. Agrawal, "A General Class of Processor Interconnection Strategies", Proc. 9th Symp. on Computer Architecture, 1982, pp.90-98

[10] D.K. Pradhan, "Fault-Tolerant Multiprocessor Link and Bus Network Architectures", IEEE Trans. Computers, Vol. C-34, No.1, Jan. 1985, pp.33-45

[11] "Transputer Reference Manual", INMOS Ltd., 1985

[12] D. May, R. Shepherd, "The Transputer Implementation of Occam", Proc. Int. Conf. 5th Computer Systems, pp.533- 541, ICOT, 1984

[13] C.A.R. Hoare, "Communicating Sequential Processes", Comm. of the ACM, Aug. 1978, vol. 21, n. 8, pp. 666- 677

[14] "Occam Programming System Manual", INMOS Ltd., July 1985

[15] "Transputer Development System Manual", INMOS Ltd., Nov. 1985

[16] G.Della Vecchia, C. Sanges, "An Optimized Broadcasting Technique for WK-Recursive Topologies", submitted to the 12th IMACS World Congress on Scientific Computation, Paris, July 18-22, 1988

```
DEF no.procs  = 16 :                -- establish the number of processors
                                    -- constituting the network

CHAN link[(no.procs+1)*4] :         -- declare the array of external channels

PLACED PAR

  PROCESSOR 0 T4                    -- allocate the front.end process onto
                                    -- the processor #0 of the network, which
                                    -- is of type T4 (Transputer T414)

    --  inside of this fold there are constant declarations for the slice of
    --  the external channel array assigned to the front.end process

    PLACE link[f.inp0]  AT link0.in :    -- map external channels on the
    PLACE link[f.inp1]  AT link1.in :    -- physical links of processor #0
    PLACE link[f.inp2]  AT link2.in :
    PLACE link[f.inp3]  AT link3.in :
    PLACE link[f.out0]  AT link0.out:
    PLACE link[f.out1]  AT link1.out:
    PLACE link[f.out2]  AT link2.out:
    PLACE link[f.out3]  AT link3.out:

    front.end (link[f.inp0],link[f.inp1],  -- load the code of the front.end
               link[f.inp2],link[f.inp3],  -- process passing the slice of
               link[f.out0],link[f.out1],  -- the external channel array
               link[f.out2],link[f.out3])

  PLACED PAR i = [0 FOR no.procs]

    --  inside of this fold there are constant declarations for the slice of
    --  the external channel array assigned to each copy of the node process

    PROCESSOR i+1 T4                    -- allocate a copy of the node process
                                        -- on the current processor, which is
                                        -- of type T4 (Transputer T414)

      PLACE link[inp0] AT link0.in :    -- map external channels on the
      PLACE link[inp1] AT link1.in :    -- physical links of the current
      PLACE link[inp2] AT link2.in :    -- processor
      PLACE link[inp3] AT link3.in :
      PLACE link[out0] AT link0.out:
      PLACE link[out1] AT link1.out:
      PLACE link[out2] AT link2.out:
      PLACE link[out3] AT link3.out:

      node (link[inp0],link[inp1],      -- load the code of the node process
            link[inp2],link[inp3],      -- passing the appropriate slice  of
            link[out0],link[out1],      -- the external channel array and the
            link[out2],link[out3],i)    -- index of the current processor
```

```
PROC node ( CHAN inp0,inp1,inp2,inp3,out0,out1,out2,out3,
            VALUE this.node.index ) =

  DEF max.length = 24 :

  VAR message[ BYTE max.length],head,length,mess.present,
      this.lev2.index, this.lev1.index, to.front.end :

  CHAN rout.chan[16], int.inp.chan[4], int.out.chan[4] :

  --  this fold contains the definition of input.process

  --  this fold contains the definition of output.process

  --  this fold contains the definition of local.process

  PAR                  -- activate in parallel instances of
                       -- output.process, input.process and
                       -- local.process

    output.process(out0,0)
    output.process(out1,1)
    output.process(out2,2)
    output.process(out3,3)

    input.process (inp0,this.node.index,0)
    input.process (inp1,this.node.index,1)
    input.process (inp2,this.node.index,2)
    input.process (inp3,this.node.index,3)

    local.process

    SKIP :
```

```
PROC input.process ( CHAN input.chan, VALUE this.node.index, gate.number) =

  VAR message[ BYTE max.length],
      dest.node, dest.lev1.index, dest.lev2.index, xor, direction,
      head,length, way.out :

  WHILE TRUE
    SEQ
      PRI ALT
        input.chan ? head                    -- message coming from the outer
          SEQ
            length :=      (head /\ #FF000000) >> 24
            BYTE.SLICE.INPUT (input.chan, message, 0, length)

        int.out.chan[gate.number] ? head     -- message generated in this node
          SEQ
            length :=       (head /\ #FF000000) >> 24
            BYTE.SLICE.INPUT (int.out.chan[gate.number], message, 0, length)

      dest.node := (head /\ #00FF0000) >> 16 -- get destination node index

      IF
        dest.node = this.node.index          -- message reached destination
          SEQ                                -- so it is sent to the
            int.inp.chan[gate.number] ! head -- local consumer process
            BYTE.SLICE.OUTPUT (int.inp.chan[gate.number], message, 0, length)

        TRUE                                 -- message must be routed
          SEQ
            dest.lev2.index := dest.node >> 2     --  get partial indexes
            dest.lev1.index := dest.node /\ 3     --  of destination node
            xor := this.lev2.index >< dest.lev2.index

            IF                                    --  find proper direction
              xor = 0
                direction := dest.lev1.index

              TRUE
                direction := dest.lev2.index

            way.out := (gate.number * 4)+direction  -- message is sent to the
            rout.chan[way.out] ! head               -- proper process which
                                                    -- handles the output
                                                    -- channel
            BYTE.SLICE.OUTPUT (rout.chan[way.out], message, 0, length)

      SKIP :


PROC output.process ( CHAN output.chan,VALUE gate.number) =

  VAR message[ BYTE max.length], head, length :

  WHILE TRUE
    ALT i = [0 FOR 4]

      rout.chan[ gate.number + (i*4)] ?  head
        SEQ
          length := (head /\ #FF000000) >> 24
          BYTE.SLICE.INPUT (rout.chan[gate.number + (i*4)],message,0,length)
          output.chan ! head
          BYTE.SLICE.OUTPUT(output.chan, message, 0, length)
          SKIP :
```

A VLSI Implementation of an Architecture for Applicative Programming

John T. O'Donnell, Timothy Bridges and Sidney W. Kitchel

Computer Science Department
Indiana University
Bloomington, Indiana 47405 USA

The Applicative Programming System Architecture contains a novel Data Structure Memory (DSM) which supports fast access operations on compact linear data structures. Several problems that arise in implementations of applicative and functional programming languages can be solved efficiently using special data representations on the DSM. Each memory word in the DSM contains a very small local processor, and there is also a tree-structured communications network within the DSM. Therefore the DSM is a massively parallel SIMD machine. This paper describes a VLSI implementation of the DSM architecture and compares its performance with implementations on a conventional sequential computer and the NASA Massively Parallel Processor.

Keywords: applicative language, functional language, SIMD, VLSI, fine grain parallelism, massively parallel processors.

1. Introduction

Applicative (or functional) programming languages have gained widespread interest because of their expressiveness, elegance and good mathematical properties [4]. Unfortunately, applicative programs are still much slower than equivalent imperative ones. There are several reasons for this. One of the problems is simply that conventional computer architectures are well matched to the implementation of imperative languages, but they give poor support for the data structure operations needed for implementing applicative languages. This paper shows how a VLSI architecture can help to solve that problem.

The applicative programming system architecture (APSA) [8, 9, 10, 12] contains a massively parallel intelligent memory unit which directly implements some of the key data structure algorithms needed by applicative languages. Each word in this data structure memory (DSM) contains a small amount of processing power, and many of the data structure algorithms cause most or all of the memory words to perform an action in parallel. In addition, the DSM contains an interconnection network, and the network's nodes also contain small processors.

Because of its massive parallelism, the DSM cannot be implemented efficiently on a single processor. Because of its fine grain organization, the DSM is also inappropriate for hardware implementation using standard chips. Fortunately, there are two promising ways to implement the DSM. The first method is to program the DSM's operations on general

purpose SIMD machines such as the Massively Parallel Processor [13] or the Connection Machine [3]. The second method is to implement the DSM directly with a custom VLSI design. Both of these methods have now been used successfully. An implementation of APSA on the MPP is already operational [11], and this paper describes the first VLSI implementation.

Section 2 gives an overview of APSA and briefly describes the kinds of data structures that it supports. Section 3 describes the architecture of the DSM and Section 4 shows how to program it. A program that illustrates the DSM's microinstruction set is given in Section 5. Section 6 then describes the VLSI implementation, and its performance is discussed in Section 7. APSA is compared with two related architectures in Section 8, and Section 9 summarizes the results.

2. Overview of APSA

It is conventional to view a memory system as an inert device which just holds data, while all computation takes place in the processor. Conventional RAM chips implement this kind of memory.

Since data structures reside in the memory while all activity takes place in the processor, operations on a data structure (such as measuring the size of a list or traversing a graph) require a large amount of communication between the processor and the memory [1]. Unfortunately, the data path between the processor and memory is only a few words wide, so any operation that affects all the components of a data structure must take place sequentially, and the operation requires time proportional to the size of the structure.

A better alternative is to design a memory that can directly perform global operations on a data structure, without sending individual words to the processor. In addition to reducing the communications bottleneck on the bus connecting the processor and memory, this approach also makes it possible for the memory to operate simultaneously on all the words in a data structure. We can do this by attaching a very small processor to each word in the memory, and including an interconnection network within the memory. This idea has been around for a long time, and it was used in several early associative machines and content-addressable parallel processors [2]. APSA also uses this approach.

Another way to think of conventional computers is that they consist of a control processor (the CPU) which issues instructions to be executed on an auxiliary processor (the memory). The memory can execute only two instructions: $x = fetch(a)$ and $store(x, a)$ where a is an address and x is a data value. APSA generalizes this view by supporting a much more powerful set of primitive operations.

The DSM supports fast access operations on compact linear data structures. For example, if all the elements of a list are stored in adjacent memory cells, then the DSM can index into the list, measure the size of the list, search for an element associatively, perform an arithmetic operation on some or all of the elements, insert a new element into the list, delete an element, etc. Furthermore, *all of these operations take unit time—their speed is a small constant independent of the size of the list.* In contrast, each of these operations requires linear time on conventional computers.

Figure 1 shows the global organization of APSA, which is just like ordinary computers except that there are two memory units instead of one. The RAM contains the control

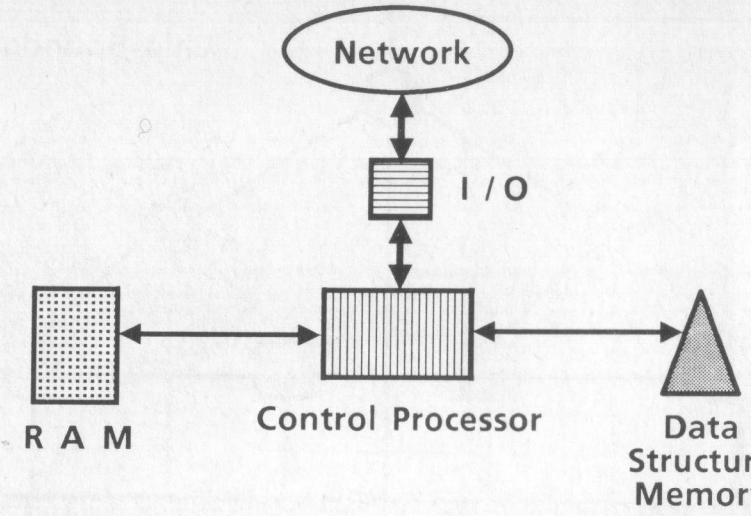


Figure 1. Global organization of APSA

processor's program and all data that can be accessed efficiently by *fetch* and *store*. The DSM stores and manipulates compact linear data structure representations (such as variable size vectors, lists, environments, aggregates, etc.).

The purpose of the DSM is to reduce the time complexity of key data structure algorithms. It does this at the cost of extra hardware complexity compared to a RAM. The next section briefly describes the organization of the DSM.

3. The data structure memory

There are two kinds of operation that must be supported by the DSM's hardware. First, there are global operations that involve communication among a set of adjacent memory words. Examples include broadcasting an instruction to all the memory cells, indexing into a list, and resolving multiple responders after an associative match. Second, there needs to be a fast way to shift a set of adjacent words in parallel in order to allow insertions or deletions in a structure.

The DSM contains two internal interconnection networks that are used to implement these two kinds of operation (Figure 2). A binary tree of combinational logic nodes implements the global operations, and data paths connecting each memory cell to its predecessor and successor permit the machine to shift all the words in a data structure in parallel. Therefore the DSM is organized as a binary tree with additional connections between adjacent leaves.

In principle, the tree nodes do not need any local processors or memory. All the necessary DSM operations can be implemented with a purely combinational tree (i.e., a

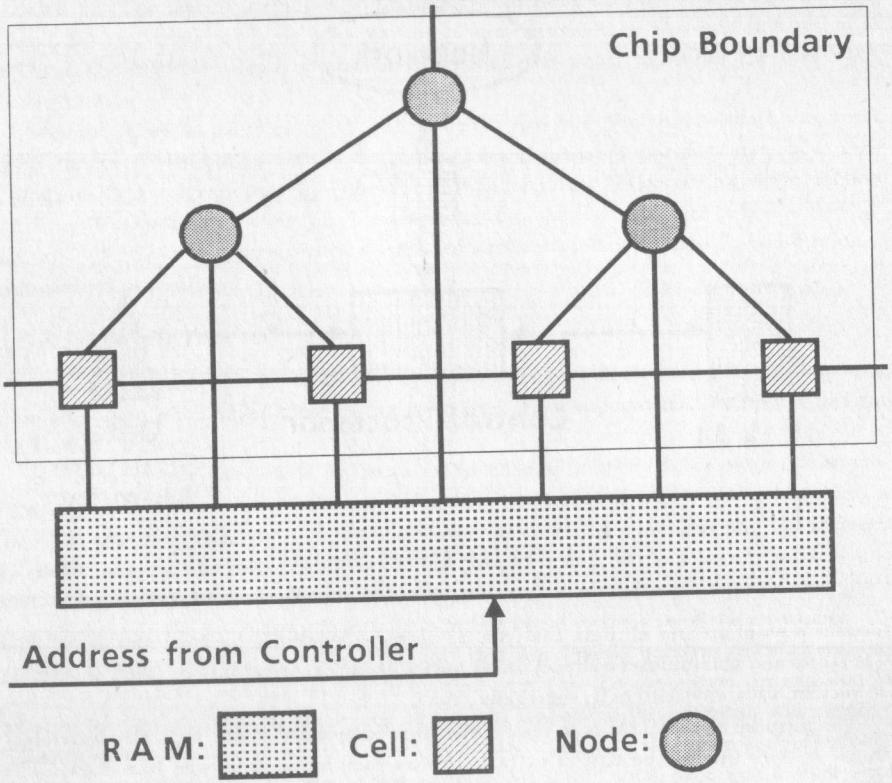


Figure 2. Organization of the Data Structure Memory (DSM)

tree where nodes contain logic gates but not flip flops). However, this organization requires wide data paths up the tree (on the order of 100 bits wide), leading to several practical problems for a VLSI implementation. It is easier to build a bit serial tree machine with 1-bit wide data paths. In order to implement some of the DSM's instructions, a bit serial machine needs to latch certain values in the tree nodes. Consequently, the tree nodes need to have local memories and local processors, just like the leaf cells.

These hardware constraints suggest a two level architecture. The high level architecture (an abstract machine) performs operations directly on the data structures. Its tree nodes are purely combinational and its data paths and registers are wide. The high level architecture would be expensive to implement in hardware (although it would be possible to do so). In contrast, the low level architecture is bit serial and its tree nodes contain memory. The low level architecture is designed to be easy to implement in hardware while still containing enough power to emulate the high level architecture. This approach, called "microprogramming", is common practice in processor design.

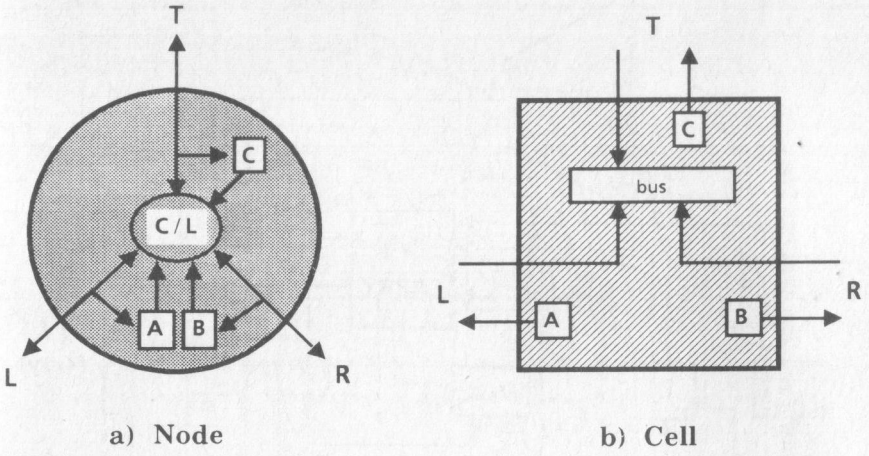


Figure 3. Processing element ports

4. The microinstruction set

This section describes the bit serial architecture of the DSM from the programmer's point of view, and the following section shows how to implement several high level data structure operations as microcode.

The purpose of the DSM is to manipulate linear data structure representations which reside in "memory cells". Each cell contains some local addressable memory (typically 128 bits or more) along with a local bit serial processor. The tree structure permits communication among the cells, but it is not used to store permanent data. The memory cells are the leaves of the tree, and we will call them "cells". The non-leaf nodes in the tree will be called "nodes".

There are two tree communication operations, called upsweep and downsweep. An upsweep causes each node to apply a logic function f to its left and right subtree inputs (li and ri respectively). The result $f(li, ri)$ is sent immediately to the top output port (to) and the input values are latched in the A and B registers. The downsweep operation is similar. Figure 3 shows the relation between the ports and registers during sweeps.

The nodes and cells have similar architectures, although they differ in the way they handle communications and arithmetic. For convenience, we use the term "processing element" or PE to refer to a node or a cell. Figures 4 and 5 show the internal architecture of a cell and node.

Each PE contains a local bit-addressable memory and four 1-bit registers called A, B, C and D. There is also a local function box which can compute an arbitrary boolean function of two or three inputs. The registers, memory port, communications ports and function box belonging to a PE are all connected on a local bus within the PE. Each PE has its own bus.

When a microinstruction requires access to the PE local memories, the control processor

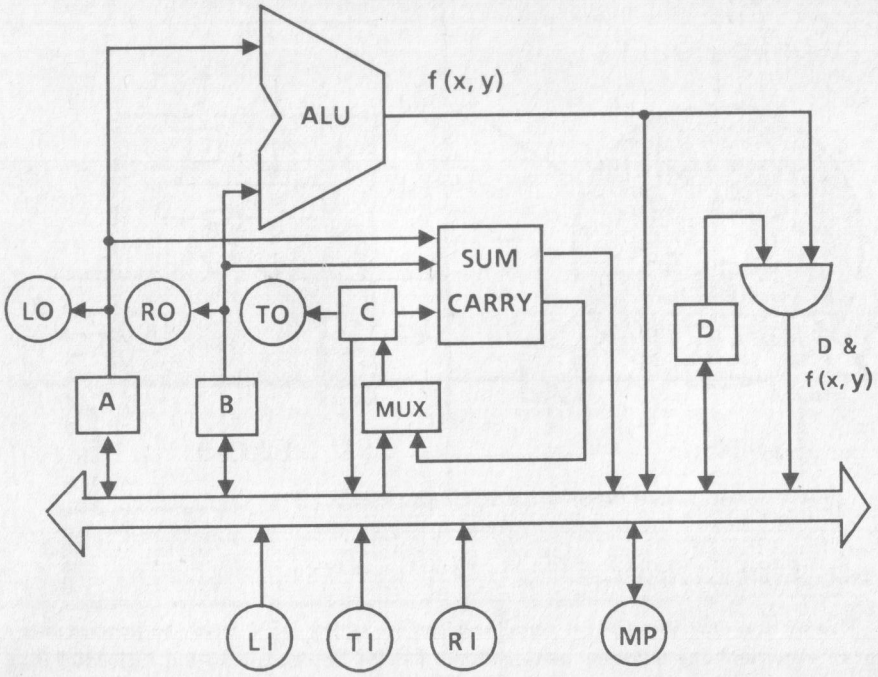


Figure 4. Cell architecture

places the address on a bus that goes to all the local memory RAMs. Therefore all PEs must access the same memory location; the hardware does not allow a PE to compute its own memory address independently of the other PEs. Relaxing this restriction would require a large increase in hardware complexity.

Each microinstruction contains a specification of which PEs are to execute it, an opcode and zero or more operands. The specification of which PEs should execute a microinstruction contains two fields. The *cnd* field (for "conditional") is a flag which may be `uncond`, meaning that the selected PEs should unconditionally execute the microinstruction; or `cond`, which specifies conditional execution depending on the value of the PE's local D register. The *petype* field specifies the class of PEs that are to execute the microinstruction. This may be `cell`, `node` or `all`.

A description of each microinstruction is given below. It is convenient to divide them into three classes: (1) the microinstructions that may be executed on any PE, (2) the microinstructions which apply only to the cells and (3) the sweep microinstructions, which involve the cells and nodes in different ways.

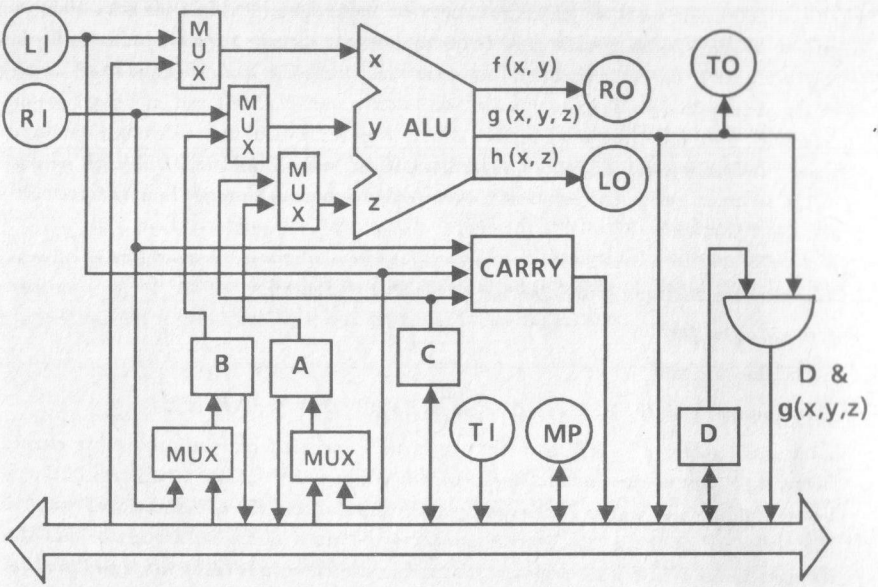


Figure 5. Node architecture

The following instructions may be executed by all PEs.

- *cnd petype* `load` *reg* := *addr*

  The bit in the PE's local memory with address *addr* is loaded into *reg*.

- *cnd petype* `store` *addr* := *reg*

  The value in *reg* is stored into the local memory at address *addr*.

- *cnd petype* `move` *reg* := *reg*

  The value in the source register (right hand side) is copied into the destination register (left hand side).

- *cnd petype* `logic` *reg* := *fcn2*

  The two-bit logic function is applied to the values in the `A` and `B` registers, and the result is copied into the destination register.

- *cnd petype* `andmask` *fcn2*

  The value `D` · *fcn2*(`A`,`B`) is placed in the `D` register.

The following instructions are only executable on the cells, so they do not need to specify *petype* .

- *cnd* `add`

  The sum of `A`, `B` and `C` is stored into `A`, while the carry is stored into `C`.

- *cnd* `neighbor` *dir*

  The direction *dir* must be either `lookleft` or `lookright`. If it is `lookleft` then each cell stores the `B` register value of its left neighbor into its own `A` register; this shifts data to the right. If *dir* is `lookright` then each cell stores the `A` register value of its right neighbor into its own `B` register, which shifts data to the left. If *cnd* is `cond` then only cells with `D=1` will store a value into a local register, but every cell makes its `A` and `B` values available to its neighbors, regardless of the value stored in `D`. The left neighbor of the leftmost cell is the left port *LP*, and the right neighbor of the rightmost cell is the right port *RP*.

The following instructions involve all the nodes and all the cells.

- *b* = `uplogic` *fcn3*

  Cells: $to \equiv \mathtt{C}$

  Nodes: $\mathtt{A} := li \quad \mathtt{B} := ri \quad \mathtt{C} := \mathrm{carry}(li,ri,\mathtt{C}) \quad to \equiv fcn3(li, ri, \mathtt{C})$

  Each node receives its left and right inputs (*li* and *ri*) from the processing elements below it. Nodes at the bottom level receive their inputs from the `C` registers of the cells below them, while higher nodes receive their inputs from the combinational outputs of the nodes below them. Each node stores its left input *li* into its `A` register and stores its right input *ri* into its `B` register. Its combinational output (sent to its top port *to*) is $fcn3(li, ri, \mathtt{C})$. The combinational output of the top port can be read by the controller and stored in a variable *b*.

- `dnlogic` *b* $fcn2_1$ $fcn2_2$

  Each node receives a combinational input *ti* from its top port and stores it in its local `C` register. The control processor supplies the top input value *b* for the top node in the tree. The two function arguments are used to compute the combinational outputs for the left and right ports respectively. The left port output is $fcn2_1(ti, \mathtt{A})$ and the right port output is $fcn2_2(ti, \mathtt{B})$. Each cell stores its top port input into its local `C` register.

The following instructions are similar to `dnlogic`, except they specify the logic function differently. While `dnlogic` allows the programmer to specify separate functions for the left and right outputs, the `dnleft` and `dnright` instructions allow a single more complex function to be used for one of the outputs, while the other output is simply connected to the top input. These instructions are useful for implementing complex instructions that treat linear data structures asymmetrically. These instructions are not implemented in the current VLSI chip.

- `dnleft` *fcn3*

  Nodes: $\mathtt{C} := ti \quad lo \equiv fcn3(\mathtt{A}, \mathtt{B}, ti) \quad ro \equiv ti$

  Cells: $\mathtt{C} := ti$

- `dnright` *fcn3*

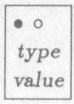
  Nodes: $\mathtt{C} := ti \quad lo \equiv ti \quad ro \equiv fcn3(\mathtt{A}, \mathtt{B}, ti)$

  Cells: $\mathtt{C} := ti$

5. APSA programming techniques

Since the DSM hardware implements very low level microinstructions, a high level algorithm should not use them directly. Instead, we define an abstract architecture with an intermediate level of data structure instructions. These instructions must be designed to be easy to use in the high level algorithm, but it must also be possible to implement them efficiently using the low level microinstructions. This section illustrates this approach by showing a typical data format for the abstract machine, along with the microcode used to implement several typical intermediate level instructions.

Each abstract machine word contains a type field, a value field and several flags. All of these fields and flags are stored in a cell's local RAM, although they will be brought into the registers (one bit at a time) whenever an instruction is manipulating them. The contents of a cell can be represented by a box with the following format:



The type field specifies whether the value field is a symbol, integer, pointer, available, etc. The "select" flag is used to locate one or more cells that are currently being used, while the "mark" flag is used for insertions and deletions. The symbol "•" indicates that the select flag is set in a cell, while the symbol "○" indicates that the mark flag is set.

A common operation is to search the memory for a cell that contains a certain value. The search instruction *match*(x) sets a select in every cell whose type field contains *sym* (indicating that the cell contains a symbol) and whose value field contains the specified symbol x. For example, suppose that the memory contains

avail	*sym* b	*sym* a	*sym* d	*sym* e	*sym* a

Now if we execute the instruction *match*(a) then the result will be

avail	*sym* b	• *sym* a	*sym* d	*sym* e	• *sym* a

The following microprogram implements the match instruction. It begins by putting the constant 1 into the A register in every cell; this is done by specifying a logic function that returns 1 on all inputs. Then the program iterates over the bits in x, clearing the A register in cells which have values that differ from x. First the loop fetches the value bit from a cell into the B register. Then it computes the new value of the A register, which is $A \cdot (B = x[i])$, where $x[i]$ denotes the current bit in x. However, there is no way to make the cells compute an expression like $B = x[i]$ directly. Instead, the program first tests the current operand bit $x[i]$, which is in the control processor, in order to decide how to calculate the new value of A: if the operand bit is 1 then $A = A \cdot B$ but if the operand bit is 0 then $A = A \cdot \neg B$. At the end of the loop the value of A is stored into the select flag location in the local memory.

```
match (x)
   uncond cell logic A f2con1
   for i := 0 step 1 until value_size
      begin
         uncond cell load B [value_addr+i]
         if x[i] = 1
            then uncond cell logic A f2and
            else uncond cell logic A f2andnot
      end
   uncond cell store select A
```

Instructions that look at a sequence of adjacent cells must usually use the tree sweep instructions. One of the simplest of these instructions is "leftmost", which clears the select flag in all cells except the leftmost cell that already had select set. This instruction provides a convenient way to deal with multiple responders after a match. For example, executing the leftmost instruction with the memory state shown above produces:

		•			
avail	*sym* b	*sym* a	*sym* d	*sym* e	*sym* a

On a conventional computer, this instruction requires time proportional to the size of the memory. However, the DSM can execute it in a constant number of microinstructions, regardless of the size of the memory. Six microinstructions are sufficient:

```
leftmost()
   uncond cell move C := A
   b = uplogic (f3or2)
   uncond node move B := A
   dnlogic b f2arg1 f2andnot
   uncond move cell B := A
   uncond cell logic A f2and
```

This algorithm consists of an upsweep which identifies which descendants of every tree node contain a cell with select set, followed by a downsweep that turns off the select flag in the right subtree of any node whose left subtree has a cell with select set.

6. The VLSI implementation

We have successfully implemented a prototype of APSA in VLSI hardware and measured its performance. The initial results are very encouraging: the DSM hardware is faster than expected, and its simplicity should enable us to construct a moderately large machine.

Since the control processor is a conventional architecture, we used a standard IBM PC AT compatible machine. The control algorithms are written in C. There were only two systems that had to be implemented in hardware: the DSM is fabricated as a VLSI chip, and the interface between the DSM and the control processor is a circuit board based on standard Programmable Logic Device (PLD) chips.

The DSM chip contains a complete tree architecture with four cells and three nodes. In addition, it contains an isolated node which makes it possible to build a larger DSM out of two smaller ones. We also placed several smaller test circuits on the chip, in order to be able to study the performance of partial circuits in case the full tree didn't work. Fortunately the entire DSM architecture worked correctly.

The nodes and cells are very simple architectures, so there is room for a number of them on a typical chip layout. We combined them with a standard "H tree" geometry [7], which leads to a relatively efficient usage of the chip area.

Before designing the cell and node circuits, one major organizational problem must be solved: how should the local PE memories be implemented? There are two alternatives: either (1) incorporate the local memories into the cell and node circuit designs, or (2) use off-chip RAM to represent the local memories. A number of issues are involved in this decision.

- Standard RAM chips have benefited from an enormous amount of refinement, and we could not hope to achieve the same memory density in a custom VLSI chip.
- Placing the local memories on the DSM chip would make the nodes and cells larger, reducing the number of processing elements that would fit.
- Standard RAM chips contain far more memory than needed for the DSM processing elements. That means we would never need to worry about running out of memory. However, building the local memories into the DSM chips would require us to decide in advance how many bits are needed. Since the purpose of this project is to support research into parallel data structures, this decision would be very difficult.
- Using separate RAM chips would increase the total chip count.
- The prototype DSM chip contains 8 processing elements (a tree with 4 cells and 3 nodes, plus an isolated node for connecting chips into bigger trees). Therefore a single RAM chip organized into 8-bit bytes could provide one bit for each processing element, and only one RAM chip would be needed for each DSM chip.

For the prototype DSM chip, it is clearly better to use separate RAMs. However, it will be necessary to reconsider this question for larger DSM designs in the future. For example, if a DSM contains 128 processing elements, then either 16 RAMs will be needed for each DSM (greatly increasing the chip count) or else 16 sequential RAM cycles will be needed to implement memory access instructions (severely slowing the machine).

Most of the DSM design effort went into laying out the cell and node architectures. Fortunately these architectures are very similar: they both consist of four ports (top, left, right and memory), four registers (A, B, C and D) and a function box, all connected on a local bus. The most important difference between a node and cell is in the ALU design.

Each PE contains a local bus used to connect the four PE registers, function box, adder, and D masking of the ALU output. The four 1-bit registers are implemented as 2-phase latches driven by ϕ_1 and ϕ_2 clocks. Access to and from the bus is provided through complementary pass transistors. The boolean function box is implemented by a multiplexor where the operands serve as the multiplexor control. The adder and D register masking of the ALU are implemented as specific combinational logic circuits.

Each PE has four I/O ports, three to provide access to one of the DSM's two interconnection networks and the fourth serving as a memory port to off-chip RAM. The data paths to and from the interconnection network ports are different in the tree nodes and the leaf cells. In the tree nodes it is sometimes necessary to look at two inputs concurrently, therefore some input ports are routed directly or through secondary buses instead of to the main bus. On the other hand, the leaf cells can only store the value of an input port into one of the registers, so all inputs can be routed directly onto the main bus.

Outputs to the interconnection networks are also different for tree nodes and leaf cells. The tree nodes' outputs are combinational outputs of a special ALU. This ALU takes three inputs, x, y and z, and produces three outputs: $f(x,y)$ goes to the right output, $g(x,y,z)$ goes to the top output, and $h(x,z)$ goes to the left output. This ALU is implemented as a tree of multiplexors. The first level of muliplexors implements the functions f and h, while the final level F combines them into the function g. Therefore the functions f, g and h are not independent, but rather $g(x,y,z) = F(f(x,y), h(x,z))$. Since some of the inputs to the ALU are combinational and the outputs to the tree network are combinational, this design implements combinational paths all the way between the root and the leaf levels of the tree. The PEs at the leaf level of the tree have no combinational outputs. Therefore the leaf cell outputs are wired directly from the A, B and C registers.

The DSM chip contains only registers, function boxes, and interconnections. All instruction decoding and control are centralized on the interface board. The DSM chips receive individual control lines that specify register loads, bus control, function selection and RAM read/write operations. This organization eliminates much redundant circuitry and reduces the area of individual PEs. As a result more PEs can be placed on a chip. The cost is that more pins are needed, but this is not a problem because the data ports are only one bit wide.

We designed the PE modules using the Berkeley 1986 VLSI Tools [14], and the circuit layouts were done using the Magic interactive VLSI editor. These designs were simulated using the ESIM event simulator and the Crystal timing analyzer. The resulting modules were tiled into an H-tree layout of the DSM using the MQUILT tiling tool. The layout specification input to this tool was generated manually since there were only 10 PE's per chip, but could easily be automated for larger layouts. The final layout contained a connected tree with seven PE's, an isolated tree node, and two isolated leaf cells. A small amount of additional circuitry was included to test the properties of the register design used in the PE's and to test the signal delay caused by pad circuitry.

The chips were then fabricated by the MOSIS fabrication facility [16]. Since this was a prototype design, we selected the cheapest technology, which was scalable CMOS P-well with 3.0 micron feature size [17]. The prototype fit onto a medium size chip of 4.6×6.4 mm and 64 pins. The scalable layout rules will allow us to fabricate this same design in denser technologies; 1.2 micron is now available through MOSIS.

Careful simulation of the design was the key to getting a working chip. The first fabrication run failed because there was an error in the standard library pad layout that we used, and we had not simulated the pads themselves. Therefore we modified the pad layout, and also learned how to make the simulator handle the pads. This meant that there were no shortcuts in the simulations. As a result, the second fabrication run was fully functional.

Size	Vax 780	MPP	VLSI
2×2	610	12	.16
4×4	2,500	20	.40
8×8	10,500	29	.64
16×16	53,000	39	.86
32×32	175,000	51	1.16
64×64	706,000	68	1.40
128×128	2,872,000	95	1.70

Table 1. 1-bit wide upsweep times (microseconds)

7. Performance results

Table 1 gives a rough comparison of the speeds of the three implementations of the DSM. The numbers give the time (in microseconds) required to perform a 1 bit wide upsweep, using the logic function $to = li$ OR ri.

The first column in Table 1 gives the "layout size" of the DSM, assuming a square H-tree layout. Therefore the total number of leaf cells is i^2 for an $i \times i$ layout.

The "Vax 780" column gives the time required by a simulator written in C running on a Vax 780 computer. This simulator requires time proportional to the number of leaf cells.

The "MPP" column gives the upsweep time for the implementation running on the Massively Parallel Processor [11]. Since the MPP assigns a separate processing element to each leaf cell and to each node at a given level in the tree, the time it takes to do the logic functions is proportional to $\log N$ where N is the memory size. However, the MPP provides only nearest-neighbor communication, so the communication time is slower than $\log N$.

The "VLSI" column gives the upsweep time for the hardware VLSI implementation. Here the total upsweep time is almost logarithmic, because the wire delay from a node to its parent is very low. We do not have enough chips to build a 16K cell machine, but we were able to measure the upsweep time by constructing an unbalanced tree. The left ports of the upper tree nodes were connected to real DSM chips, while the right ports were connected to constant logic values. This does not produce a usable DSM, but it does allow fairly realistic timing measurements. However, very large fabrications of the DSM will probably have additional delays due to longer wires and connections between boards.

Table 1 makes it dramatically clear that implementing APSA on a von Neumann architecture is pointless, while true parallel implementations—using either general SIMD machines or VLSI—are fast enough to be practical. The VLSI implementation of the DSM is slower than a RAM by a constant factor, but it provides a linear speedup for algorithms that traverse linear data structures.

8. Comparison with related work

APSA is loosely related to two other fine-grain parallel implementations of symbolic languages: the FFP Machine (or FFPM) and Connection Machine Lisp. This section discusses the similarities and differences among these three systems.

The FFP Machine [5, 6] is a massively parallel implementation of the FFP language [1]. The FFPM and APSA were developed independently at about the same time. Both of these architectures contain a linear array of processors (called cells in APSA and L-cells in the FFPM) with an additional binary tree of processors (called nodes in APSA and T-cells in FFPM). Numerous variations on tree architectures have been studied for a long time, but APSA and FFPM use the general organization in novel ways.

APSA and FFPM both contain a tree structure and both are intended for implementing functional programming languages, but these similarities are rather superficial. They differ in several fundamental aspects:

- The FFPM is restricted to string reduction because it cannot represent pointers. In contrast, APSA was designed from the beginning to handle pointers well. APSA is unique in its ability to perform both string reduction and graph reduction efficiently. The primary problem with the FFPM is that string reduction may cause enormous communication overhead. This happens when two expressions being evaluated in parallel need access to the same data structure. With string reduction separate copies of the data structure must be made for each evaluation, while graph reduction allows the data structure to be shared via pointers.
- The FFPM is an MIMD architecture which runs a parallel algorithm distributed through its cells, while APSA is an SIMD architecture which runs a sequential algorithm in its control processor. In this respect, APSA is like the Massively Parallel Processor and the Connection Machine, and it is quite different from FFPM.
- The FFPM is an asynchronous machine, while APSA is synchronous.
- The abstract version of APSA can be implemented on standard SIMD machines (and this has already been done), but the FFPM cannot.
- The hardware complexity of the tree nodes and cells is much smaller in APSA than in the FFPM (by approximately 2 orders of magnitude).

Connection Machine Lisp [15], like most programming languages for SIMD architectures, is a sequential language that supports parallel operations on data structures. The primary data structure in CM Lisp is the *xapping*, which is an unordered set of index-value pairs. Indices and values may be arbitrary Lisp objects. CM Lisp contains several interesting operators that map functions over the elements of a xapping in parallel, and there are several mechanisms for controlling this parallelism.

APSA and the Connection Machine illustrate different tradeoffs between hardware complexity and programming style. The tree structure in APSA is able to perform parallel operations on the objects in a linear data structure representation, but it does not have enough communication paths to handle non-linear representations. The Connection Machine does not have this restriction because of its powerful routing network. However, the Connection Machine requires much more complex hardware than APSA.

9. Conclusion

APSA contains a massively parallel intelligent data structure memory that supports a number of powerful operations on linear data structure representations. These operations

are especially useful in implementing applicative programming languages and for symbolic processing.

The data structure memory has a very regular architecture which is well suited for VLSI implementation. We have constructed a prototype chip and measured its performance. The VLSI machine implements the critical tree sweep operations approximately 50 times faster than an implementation on the general purpose Massively Parallel Processor. A sequential implementation on a Vax 780 runs more than a million times slower.

The usefulness of APSA will depend on whether the algorithmic speedup provided by the DSM can overcome the cost of its hardware and its slower performance. The initial results look very encouraging, since several interesting algorithms run faster on APSA by $O(n)$ where n is the problem size. Furthermore, the organization of the DSM is very well suited for denser circuit fabrication techniques such as wafer-scale integration.

Acknowledgements

NSF and DARPA supported the MOSIS VLSI fabrication facility. The staffs at ISI and MOSIS helped us get started, David Winkel gave useful advice on design problems, and Robert Wehrmeister supported our design tools.

References

1. John Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", *Communications of the ACM*, Vol. 21, August 1978, pp. 613–641.
2. Caxton C. Foster, *Content Addressable Parallel Processors*, New York: Van Nostrand Reinhold Co., 1976.
3. W. Daniel Hillis, *The Connection Machine*, Cambridge: The MIT Press, 1985.
4. John Hughes, "Why functional programming matters", Report 16, Programming Methodology Group, Chalmers University of Technology, Göteborg Sweden, Nov. 1984.
5. Gyula Magò, "A network of microprocessors to execute reduction languages", *Int. Journal of Computer and Information Sciences*, Vol. 8 No. 5 and Vol. 8 No. 6, 1979.
6. Gyula Magò and David Middleton, "The FFP Machine—a progress report", *Proc. International Workshop on High-Level Language Computer Architecture*, May 1984, pp. 5.13–5.25.
7. Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Reading, Mass.: Addison-Wesley, 1980.
8. John T. O'Donnell, *A Systolic Associative LISP Computer Architecture with Incremental Parallel Storage Management*, Technical Report 81-5, Computer Science Department, University of Iowa, Iowa City, 1981.
9. John T. O'Donnell, "An architecture that efficiently updates associative aggregates in applicative programming languages", *1985 IFIP Symposium on Functional Programming Languages and Computer Architecture*, Springer-Verlag (Sept. 1985) 164–189.
10. John T. O'Donnell, "An efficient architecture for implementing sparse array variables," *Proceedings of the Twenty-third Allerton Conference on Communication, Control and*

Computing, pp. 986–995, Coordinated Science Laboratory, University of Illinois, October, 1985.

11. John T. O'Donnell, "Parallel VLSI architecture emulation and the organization of APSA/MPP," *Proceedings of the First Symposium on the Frontiers of Massively Parallel Scientific Computation*, NASA Goddard Space Flight Center, Sept. 1986.
12. John T. O'Donnell, "Finely grained parallelism in an applicative architecture," *Proceedings of the Workshop on Future Directions in Computer Architecture and Software*, pp. 372–374, Army Research Office, May, 1986.
13. Jerry L. Potter, ed., *The Massively Parallel Processor*, Cambridge: The MIT Press, 1985.
14. Walter S. Scott, Robert N. Mayo, Gordon Hamachi, and John K. Ousterhout, editors, *1986 VLSI Tools: Still More Works by the Original Artists*, Computer Science Division, University of California, Berkeley, December, 1986.
15. Guy L. Steele, Jr. and W. Daniel Hillis, "Connection Machine LISP: fine-grained parallel symbolic processing," pp. 279–297, *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming.*
16. USC Information Sciences Institute, *MOSIS: MOS Implementation System, User's Manual*, 1986.
17. Neil Weste and Kamran Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*, Reading, Mass.: Addison-Wesley, 1985.

Logic Programs Directly Processed in a Network of Content Addressable Memories

Armand de CALLATAY

IBM European Systems Research Institute
La Hulpe, Belgium

and Unité d'Informatique,
Université de Louvain, Belgium

A task-oriented controller to manage learning robots was described in Callataÿ (1986). The main processing element is a phasic categorizer, the output of which is an instantiated symbol transmitted to many other categorizers. Categorizers are the nodes of a directed circular network processor system (DNPS). The impulses (instantiated symbols) concurrently flow within the DNPS loops. The system is radically different from conventional computers and parallel processors. Memory consists of relationships which are stored in the processing elements: these are "recording content addressable memories" (RCAM) which can do only one machine instruction type: "association for production rule processing". "Arithmetic and logic units" and "addressable memories" are suppressed. A well-designed DNPS can resolve complex problems, like some rule based or logic programs. Predicates do not appear in the compiled programs. They are replaced by instantiable keywords. Production rules are represented like documents described with keywords and are processed by an information retrieval algorithm. Processing may continue even when no rule matches because analogies may be discovered by partial matching and produce a relevant pseudo-logical solution from incomplete knowledge. The active working memory is matched in parallel with the whole rule base. The DNPS controller is suggested as the central part of a brain model.

It is natural to represent a problem and describe resolution in logic notation or in Prolog programs. These programs are implemented on von Neumann machines with a machine language quite different from the simple source notation. This paper describes a processor architecture in which logic programs are directly implemented. It proposes a computer type radically different from usual computers: the system has no arithmetic and logic units.

Parallelism: Today's parallel processors do not improve the above notation-implementation gap. They are usually made of CPUs connected by a network to the memory sets. They avoid the processor-memory single bus *(the von Neumann bottleneck)*. The connection machine[1] (Hillis, 1985) has on every chip many processors computing and also routing messages: thus the network is replaced by dynamic message routers. The parallel processor presented here has a one-way communication network, not between von-Neumann processors, but between recording content addressable memories (RCAM).

Knowledge processing: The directed network processor system (DNPS) maps knowledge representation to hardware physical connections and goal resolution to circuit processing. Pattern matching, the main operation in production rule systems and in logic programming, is made with one instruction in RCAM.

([1]) - "Connection Machine" is a registered trade mark of "Thinking Machine".

Paper outline: The first part provides a simplified formal description of the processor. The second part shows how the DNPS computations might be understood. The third part describes sample operations: the control of simple robots, and the execution of production rule systems and logic programs.

The aim is to describe a processor architecture for task-dependent controllers of robots. My engineering method is to study analogies with known devices and theories to guess whether a prototype will work or not. A formal logical justification of the processor computation has not been found and may not exist. Justification should come from successful experimentation. Some prototypes were described in Callataÿ (1986).

1) Directed network processor systems (DNPS)

The following defines the objects and processes used in the kernel of the DNPS machines. The peripheral components - sensory captors, feature extractors, servomechanisms, rewarding functions, robot hardware, or interfaces to an intelligent end-user - are a necessary external environment, but only their interface is described here.

Primitive concepts: A directed network processor system (DNPS) is described in the following and illustrated in figure 2. The objects (cells, groups and links) are permanent, but the states (activation, weights and control signals) are values which can be changed or can spontaneously decay in every period. The following entities are defined:

- *Cell:* each cell has an activity state and a habituation state.
- *Group (or categorizer):* each group has a type, contains a list of cells, and receives control signals - synchronization, reward, hold, enable. The group state is identified by the most activated cell.
- *Link:* each link has a direction from an origin to a destination cell. It includes a switch with a variable conductance - a *synapse* having a weight. An impulse from a cell is distributed into all its links, i.e. its *axon*. All cells of a group have the same type of output links. The most frequent type of switch is the *latch*. Latches are initially open, but may be irreversibly closed in one operation. Cell activation increases with each received impulse in proportion to the link conductance. A set of links between two groups is a *cable*. Other links have a given permanent conductance. A *group enabling cell* has an axon with such links to all cells of a group.

Each cell - whose links are inputs to other cells - has the properties of a *keyword* (unique identification, instantiability, automatic parallel distribution of its state to other groups). A group state is an instantiated keyword. In an inverted file organization and in RCAM, a keyword corresponds to a list of pointers to the documents containing it. With fuzzy document information retrieval, keywords in queries and document description have a weight.

Long term memory, permanent relational memory, is best understood as pointers between cells. *Short term memory*, transient state memory, is best understood as activated cell states.

Keyword notation is chosen for sets of states - *patterns* - because:

- Few cells are ON.
- Cells are not ordered within a group.
- Inactivated cells are masked in the processing.
- Recognition is made as in document retrieval, with partial matching and sorting based on the weighted summation of keywords per document. This recognition is a *categorization* when only the best document is selected. Message sets describing ambiguous entities may also be processed (Callataÿ, 1987a, 1987c).

Recording Content Addressable Memory (RCAM): An RCAM is represented in figure 1. The RCAM hardware for symbolic processor has *latches* which are all-or-none and irreversible (*additive memory* feature). The content addressable memories with fully

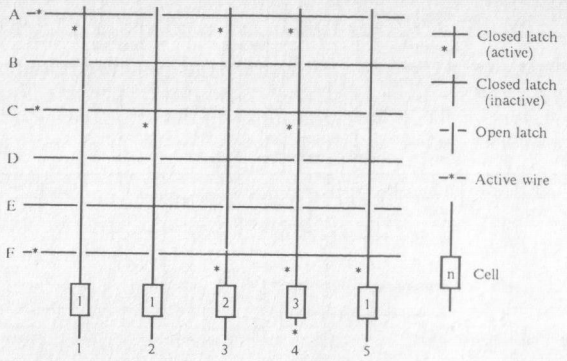


Figure 1. Information retrieval in a content addressable memory: *This diagram is well known: it was used to describe the McCulloch-Pitt's cell, the associative memory, the Rosenblatt's perceptron, the Marr's neural model and the Willshaw's "non-holographic" memory. Here the activated input wires (in A, C and F) carry keywords. Each cell counts the number of impulses received through closed latches (crosses with an *). The conduction of closed latches may be weighted, allowing information retrieval with keywords having weights in document and query. Keyword intensity may be the frequency of input impulses. The most activated cell (cell 4) is instantiated (i.e. fires).*

logical matching of my first model (Callataÿ, 1969a) was replaced by systems allowing matching on incomplete information, because these have better capabilities. The similarity based matching of information retrieval permits retrieving data with incomplete knowledge. Similar functions are done in memory-based reasoning (Stanfill and Waltz, 1987).

RCAMs associate data with AND conjunction. When RCAM cells have the same output, they may be combined with OR conjunction. OR is sometimes replaced by AND in approximate document retrieval.

In an extended RCAM, the closed latches have a slightly modifiable weight. This last RCAM executes an information retrieval algorithm with partial matching on fuzzy data.

An RCAM can be simulated by a microcomputer. A DNPS is built by interconnecting the microprocessor chips with a network. Task-oriented DNPS processors can be made with each cable replaced by a branching wire for one-way synchronized message communication (Callataÿ, 1986).

Categorizer function. In the simple model, an arbitration mechanism is part of each group (RCAM), therefore a processing rule is: *"only one cell per group can fire at a time"*. Each group is a classifier or categorizer. The output symbol of the classifier is called the *group state*. A DNPS is a directed network of cables whose nodes are classifiers. Extended versions of the model permit conflicting outputs with associated weights (Callataÿ, 1986, 1987a, 1987c).

DNPS processing waves: The abstract cycles correspond to those defined in logic circuits. Operations are usually represented with equal time intervals, but central commands may select different clocks to vary the interval duration. The processing is *"rhythmic"*. *"Periodic processing"* was the main feature of my first paper (Callataÿ, 1969a).

The following are structural definitions for DNPS networks:

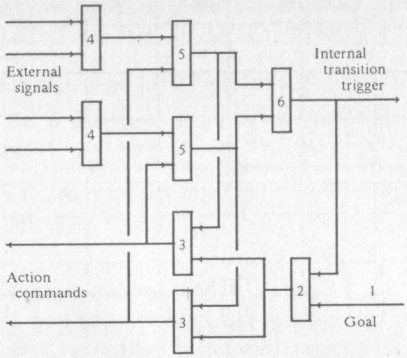


***Figure 2. Directed network processor system** (DNPS): Each box represents an RCAM. The chain numbers (the levels) are represented in the boxes. 1 is the top floor; 4 is the bottom floor. 4, 5 and 6 are the bottom-up network. 1, 2 and 3 are the top-down network. 2, 3 and 5 are the floor 2 (a functional cortical column). 6 stores the top event. The changes of state occurring in the activated nodes can be viewed as a concurrent synchronized processing wave travelling in the DNPS, a physical wave corresponding to the abstract control flow occurring in the flow charts of parallel programs. The system operates in sequential phases synchronized by external event signals or interval timers. State changes in cells and groups occur within a cycle but may persist for one or more periods. A node has no more than one active cycle per period (well-formed DNPS). All DNPS loops can be processed at each period.*

- *A chain number:* is a number n assigned to each group such that: $n_i \geq n_j + delay_j$ when there is a *link(Ci, Cj)*
- The chain is *tight* when the delay is one cycle.
- *A chain* is a list $[C_1, .. C_i, C_{i+1} ...]$ such that there exists a $link(C_i, C_{i+1})$ for every i. A list of chain numbers (levels) is associated: *[1, 2, 3, ...]* (numbers are immediate successors in *tight* chains).
- *A processing loop* is a processing chain enumerated until a repetition of a group identifier occurs.
- *The maximum loop* is the processing loop with the greatest tight chain before any chain number is encountered a second time.
- The *period* is the time needed for the maximum loop execution.
- *A well-formed DNPS* is a DNPS network such that:
  - A node is never activated more than once per period.
  - No node is its own subsequent node,
  - All nodes can be activated at each period,
- *A pure DNPS (logical DNPS)* is a DNPS where all associative processors are RCAMs with permanent threshold cells and pure latches (no adaptive weights).
- *A DNPS subset* is made of one or some groups. It may correspond to the Minsky's agency (1986).
- *An extended DNPS* is a well-formed DNPS with additional devices such as registers, stacks, state loaders and state transferring devices. The additional devices do not interfere directly with the DNPS processing but act to start and control a periodic process.

Parallelism in rule resolution applies at three levels (Callataÿ, 1981a):

- *Cell:* Each active link adds a quantum to the cell potential (neurophysiological processes are more complex: see chapter 4 in Callataÿ, 1986).
- *Group:* All cells are concurrently counting. Arbitration is partly done in parallel.

- *DNPS:* All groups with the same chain number work simultaneously.

Some of these concurrencies are described in Blelloch (1986).

Robot control with methods of qualitative physics: The following simple robot illustrates the operations of a DNPS controller. This robot must move things to a given position. Physical measurements (position, speed) are converted into qualitative symbols, e.g. *far left, near left, near right, far right* and *fast left, slow left, slow right, fast right.* An engine applies one of the following forces to the thing: *strong left, weak left, weak right, strong right.* A rule is applied for a time interval (Callataÿ, 1969a), e.g.

When movement must be stopped in the center
and the position is far left
and the speed is slow right
then the force to apply is weak right.

With degrees of membership, a measurement can include two symbols: e.g. 40% far left, and 60% near left. Then several rules are applied with various coefficients and the rule actions generate commands from which a precise numerical value is computed. The fuzzy set theory gives the formulas for rule combination. Such robots were recently demonstrated in Tokyo, e.g. to maintain a vertical pole in equilibrium on a moving cart.

Qualitative physics is described with rules, whose number could be increased when the logical consistency is kept. With many rules, AI researchers hope to manage more complex behavior than with conventional programs or formulas. Redundant rules in separate devices can enhance the reliability.

A global robot system uses many computational principles. The central symbolic DNPS processing is integrated with several cybernetic systems based on analog computation, e.g. relaxation computation and Fourier transforms in feature extractors, set-point regulation in servo-mechanisms, and perceptron computation in the choice of intentions.

Robot controller definitions: The following definitions have meaning for a DNPS controlling a robot, and by extension for systems allowing problem resolution.

- *Top and bottom:* By convention a group is labelled *top.* The top node has the chain number 1. Another group is labelled *bottom.* In the following, the top corresponds to the main goal or intention, and the bottom to the external input/output.
- *Derived tops and bottoms:* Because of network parallelism, several nodes must have the same chain number.
- *Top-down network (intention network):* chains with increasing chain numbers between top and bottom nodes.
- *Bottom-up network (sensation network):* chains with increasing chain numbers between bottom and top nodes.
- *Floors* are top-down level numbers which apply also to the corresponding bottom-up nodes in the functional cortical columns (figure 3 and Callataÿ, 1986).
- The *Enable commands* are transmitted by links enabling groups or areas. They may circumscribe the context in which a stored event is a valid model of the present situation.
- *Inner and outer networks:* When a group enables an area, it is *inner* to this area. The area is *outer* to this group. A hierarchy from an innermost group is built in that way (several DNPS hierarchies operate simultaneously: Callataÿ, 1986).

Primitive DNPS networks: The above elements are the building blocks of a processor architecture. The suggested DNPS system *is not a general purpose machine.* It cannot be programmed with imperative commands. The depth of recursion at each period is limited by hardware.

Five typical DNPS were described in Callataÿ (1986):

- The finite state machine (FSM) made of an RCAM and a register.
- The production rule system (interconnected FSM).

- The context free parser (push-down automaton made of an RCAM and a stack).
- The cascaded augmented transition network for extended parsing (interconnected parsers).
- The bidirectional hierarchical network for clause resolution (e.g. figures 2 & 3).

Traditional computers and Lisp machines already have FSM and push-down automaton. These FSMs are used to manage ALU and to command other circuits controlling addressable memory. What is new is that an FSM in a DNPS directly processes the application program, and no more the internal commands of the machine language. DNPS I/O must be done with symbols. Thus DNPSs need attached devices producing and consuming I/O symbols.

2) Meaning of DNPS processing

A modified Frege's ontology permits logic to be applied to the DNPS circuits: *Everything is either a situation or an object or a function.* Thus a direct mapping to the DNPS model gives:

- The *objects* are cells (components, devices),
- The *situations* are configurations of cell states at the same moment (all cells of a well-formed DNPS have only one ON/OFF state during a period). An ON state is called a *condition* and identified by an instantiated *keyword.* A situation is thus a set of instantiable keywords.
- *Functions* are replaced by *productions* processed in RCAMs.

Contrary to classical logic, where predicates are true or false outside the time dimension, robots have changing states and must be described by a temporal logic based on productions. In classical logic, a *function* can only change a state from the unknown state to the known state and is always applicable. By contrast, a *production* can change the states from true to false or inversely.

Many events are possible now, but not simultaneously. Contrary to *functions, productions* are not necessarily executed: they may *fire,* i.e. change their binary state, be *instantiated.* An instantiation results from a decision with multiple choices.

Each object is a symbol. An entity analogous to the DNPS symbol is a Lisp pointer, but characters, integers, "quotes", "lambda" and "nil" symbols are not primitive entities in DNPS. I explain later that the activation of an object should be understood as *"The predicate corresponding to the symbol is true in the present environment (implicit variables, context and modality)".* When a program is constrained to manipulate only pointers, it can still do non-trivial tasks, as shown by many Lisp programs.

Meaningful information can only be stored in some dedicated parts of the DNPS hardware. The meaning of a DNPS symbol comes from the wired-in references to the external world interface. There are many potential wired-in relations: only a few ones are selected by the method of *learning from experience* (latch closures). As only few connections can be learned at each period, it is common that a complex meaning cannot be stored in a "young" machine. Training must be progressive.

Difficulties of explaining DNPS with circuit logic: The following shows why the traditional logic methods of computer studies may not be appropriate as the foundations of DNPS computing and suggests other paths of research which logicians should clarify.

The DNPS are not general purpose machines and are not equivalent to a Turing machine. DNPS are not fully programmable. The main program is built in hardware; the rest may be learned or loaded in the latches (this last operation is not possible in the corresponding brain model). For instance, when DNPSs are black boxes helping to understand spoken or handwritten natural language, these DNPSs cannot be programmed to solve arithmetic problems.

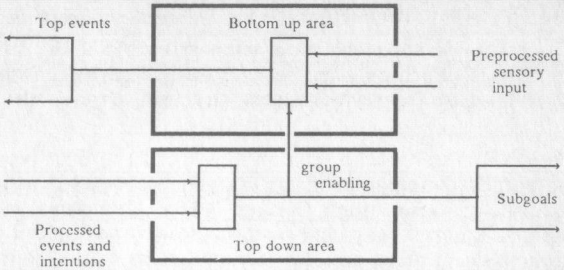


Figure 3. Simplified functional cortical column: *A system made with a hierarchy of the above DNPS subsets can resolve problems of deductive logic.*

The elements of conventional logic are permanent individualized objects which are not those processed by DNPS. The elements processed by a robot controller are the transient phenomena acting on the sensory captors. Each DNPS cell is an individual permanent element, but a cell does not correspond to an object of the external world, like in the abstract meaning given by an user to the objects of his conventional program. The sensory data from "a sea wave in movement" are a primary situation.

The elements on which the computations are done are situations, and each situation has a unique representation. Computations are made according to the functional calculus of order Omega based on the following definition: *X = Y stands for: "for all predicates F, F(X) implies F(Y) and conversely".* As all individuals having the same properties are equal, only *generic* or *function* types exist. Applicative languages have only functions: they suppress constants by replacing them with functions independent of their arguments.

A DNPS cannot represent two individuals having the same properties. No more than one instance per concept can be active, but a concept may have "quantity" as a property slot. The spontaneous DNPS computation can only process generic entities: *actor types* as opposed to *individuals*. An actor type is represented in a spiny neuron in my brain model (the human brain has more than 10^9 such cells).

Paradoxes may not appear in DNPS because the RCAM input domain is different from the output domain. In the DNPS processing, data are external signals recognizing stored knowledge and generating action commands under the management of goals, another kind of external signal. The above processing phase has no loop and is recorded as an event describing a real behavior of the physical world. The internal state, resulting from the event, partially becomes the initial internal state of the next period. The possibility of having paradoxes in this last loop should not prevent the experimental study of such operations.

Events and episodes: When *'event'* is not used for the *occurrence* of a state transformation, it designates the *trace* of a state transformation. *A stored event is here a production rule* (see later). The operations resulting of memory-based reasoning (Stanfill and Waltz, 1987) seem equivalent to production rule firings.

When the conditions, the causes, of a state change occur, this potential event instantiates the RCAM cell associating these conditions.

A hierarchy of events can be defined. An instantiated higher level event means that the association of lower level events causing it are present. With categorization on partial patterns, many events may have the same top cell.

Muggleton (1987) described several operators to learn how to structure a hierarchy. He noted that a teacher (an oracle) might not be required when many examples are

available. It would be interesting to study whether similar methods can spontaneously assign the recording cells of intermediate levels.

Monotonic logic remains applicable in a temporal world if the stored events can compute the state changes. Such an *event calculus* formalized by Kowalski (1986) uses an *additive historical database* from which states, structural parts of the events, may be deduced.

Episodes are top events stored with relationships specific to their occurrence in time. These episodes are needed to recognize that a situation occurred twice. Remembering episodes is a prerequisite of consciousness (Klages, 1926). New events may be *chunks* of past events. The chunks improve reasoning performance (Newell). Rau (1987) has built systems having an episode memory and two conceptual memories called abstract and semantic.

When a stored event is reactivated, it generates a state which can be used as an intention or as a fictitious initial state. Remembering is normally initiated by a clue activating a past episode. Forgetting is frequently due to wrong or missing clues. With the above extended DNPS, states in middle floors (*simulus* as defined by Minsky, 1986) may be reloaded. A simulus may be assembled from many events, providing a mechanism for imagination and creativity.

Things: Because the objects of the logic processor are mapped to classes, permanent and individual things cannot be represented, but can be discovered by inference. They are deduced from regularities observed in the chaining of episodes. Episodes concerning a permanent thing are correlated.

Things are understood by their analogy with groups of production rules: both have a behavior. In men, things are understood as: *"Egos projected in the world"* (Klages, 1926). This organization explains some psychological experiments and avoids some problems of non-monotonic logic. The spontaneous understanding of the world, which was observed in primitive men, is not based on permanent things, but on functions outside time and space, according to the controversial interpretation of Levy Bruhl (1922).

Memory: Several types of memory are included in the model. Here are some definitions:

- A *continuous* memory stays unchanged until a new *Write* command is executed. The short term memory (STM) uses this method, but its *Write* erases the other states of a cell group, area or level.
- A *transient* memory spontaneously decays with time or disappears at the period end, unless it is kept by a *Hold* control command.
- A *permanent adaptive* memory stays unchanged until an *Adaptation* command is given.
- A *permanent irreversible* memory has only *Set* commands (no *Erase*).
- *Recording* is an irreversible modification. This write-once/read-many (WORM) method creates an *additive database*. Recording changes a weight from 0 to 1, i.e. closes a latch. Recording is here the main learning process: communicating cells which are activated at the same period are linked together (Callataÿ, 1969a) according to the Hebb back-propagation method. This rule can be extended for multilayered networks (Callataÿ, 1969a, 1986) and is applied for enabling recording in functional cortical columns. Recorded events allow restoring a state, as done in the K-lines of Minsky (1977, 1979, 1986). Recording permits solving the *duplication problem* of Minsky (1986). A new event, similar to an old one, is recorded as a new, different episode. Then the focussed parts of episodes can be compared, and differences of details discovered.
- *Adaptive learning* (or learning) is a permanent adaptive modification. A weight change in perceptrons and adaptive neural networks is progressively made in connections serving many situations; therefore it simultaneously produces learning of some

relationships and forgetting of others (this feature is hardly compatible with the robustness of human long term memory).

Recording is done according to the single assignment rule (Callataÿ, 1981a). This applicative method makes non-destructive assignment. The stored rules are true (or approximately true) because they are learned by experience in a repetitive physical world. Therefore the deductive process can only find accurate actions because the rules remain true.

In DNPS with membership grades, adaptive learning may be superimposed on circuits already recorded. The adaptive changes caused by learning are negligible in comparison of the recorded ones, but are significant to choose between possible solutions. During a period, the activated latches define a small network subset where the principles of adaptive neural net may improve learning of the best choice.

The rules *"Perception → Intention"* are learned by perceptrons with adaptive learning (perception represents a group of events). The above rule replaces the behaviorism system: *"Stimulus → Response"* (Callataÿ, 1986).

Feature extractors cannot learn in my model, because these modifications would change the meaning of symbolic features and spoil the subsequent symbolic processing.

Memorization (latch closure, weight or threshold changes) is conventionally said to occur at the period end.

Recognizers (interneurons or feature extractors) and memorizers can be separated (Callataÿ, 1986; Minsky, 1986) or combined in an RCAM.

3) DNPS operations

Learning methods: What is learned are productions. These are *hypothetical statements* which are always true, and are applicable or not depending on the environment. The Stoic *resolution method* with *propositions* was formalized as the *production system* (Post, 1943, described in Minsky (1967) who introduced the concept in computer science) and developed in the psychological model of Newell (1973). The use of variables in the propositions enhances the role of rules: instead of being only repetitive, they can generate novel solutions.

Recording the state of a robot controller before and after a command permits finding the action producing an intended result. The rule:

State & Action → Result

is stored as the dual one:

State & Intention → Action.

The *General Problem Solver* of Newell and Simon (1972) is based on the more detailed rule (as shown by Minsky, 1986):

Situation and Goal → Difference → Agent.

These productions are added to the storage of the DNPS groups during lifetime. The *before state* is held in a buffer until the result of the action is known for simultaneous recording in RCAM. Trials are recorded as a structure of complex rules forming a whole (a complex event or episode which is globally true).

Behavior rules: A theory of DNPS robots must make assumptions about the environment. A *perfectly repetitive external world* is defined with the following equations (note that *perception* is here a concept simpler than in psychological perception):

Given known patterns S (sensation) and D (drive) and corresponding cells P (perception) and I (intention), previously learned rules allow determining the pattern C (command) by the following associations:

$S \rightarrow P$ *(recorded)*
$P \& D \rightarrow I$ *(adaptive, learned by reward)*
$P \& I \rightarrow C$ *(recorded event: I is changed after an unsuccessful trial)*

such that the physical world response P′ to C gives:
$P' \rightarrow I$ *(built-in)*

In a repetitive world, a robot recording the above rules repeats previous global behaviors, if any. Such a model has the interesting logical properties of a perfect finite state machine, but it is inadequate and inefficient. Improvements come from device enhancements:

- Events with the same outcome are grouped.
- Noisy input is progressively reduced by habituation (reversible learning).
- Rules apply in similar cases (partial fuzzy matching).

Habituation increases the firing threshold of a cell which was activated by an event producing boring results (neither rewarding nor threatening). Habituation suppresses irrelevant data (noise).

Multilevel DNPS networks: The main enhancement is the following: *intermediate nodes are inserted* transforming the FSM into a DNPS. Then the combinatorial explosion problem is replaced by a categorization problem which may be solved with ad-hoc classification algorithms (Callataÿ, 1986). The transformation permits introducing more improvements:

- Complex rules are split into trees of simple rules.
- Data irrelevant to a local context can be eliminated because knowledge can be partitioned.

The logical consistency of the above FSM is lost, but a workable consistency possibility may subsist depending of several factors (Callataÿ, 1986, 1987b). Experiments with these hypotheses may show whether an outside world with frequently repetitive responses can be managed. A well-formed DNPS is logic only when there is a purely logic solution, i.e. when the knowledge base contains fully matchable rules and relevant categorization.

Here the logical axioms of learning robots are not simple generalized rules, but whole events. Inferences are done with partial matching in this large knowledge base of episodes recorded in a lifetime. With the above assumptions, a purely logical solution to a problem is infrequent in animal real life. Adult men can use logic because they can abstract problems.

Having several layers of imbedded processors which are transiently self-sufficient is required for achieving high levels of behavioral performance according to the Papert's principle (in Minsky, 1986). The system permits speculations on the Piaget's development stages (Callataÿ, 1986). In the above hierarchy, the high level commands select the variables and rule sets processed in a low level by enabling areas of STM and RCAM (or equivalent processes). DNPS have several types of hierarchies (Callataÿ, 1986).

Expert systems and DNPS: In the following, I explore whether DNPS can resolve problems like expert systems. The production rule algorithm is easy to implement with a DNPS loop between registers and RCAMs (Callataÿ, 1986). Complex inference engines use blackboards, partitioned domains, hierarchies of problem solving systems, complex rules with multiple variables, and logical deduction methods, e.g. in Prolog interpreters.

Working memory, context and attention focus: When the results are stacked in STM, the circuits are again available to process the same types of elements with different variables according to the *recursion principle* of Minsky (1986). The active variables are the present external signals, or the present "simulus". This subset of the external world, *a*

world slice or *a context,* is selected by the attention focus. The present real world slice in the bottom-up DNPS becomes one of the possible microworlds instantiable in the top-down DNPS. The knowledge data (ascending and descending *nemes* of Minsky, 1986) circulate within the DNPS loops to progressively perceive (Callataÿ, 1986).

As the focussed world frequently changes, what is instantiated in the brain is a sequence of world slices with *asserted* and *intentional* modes. A similar method is used in expert systems: the data of different world slices are stored in different areas of a *blackboard* where, at each period, a selected subset (context, goal and modality) is matched with the rules (HayesRoth, 1985). In my model, blackboard areas are activated by focussing on a limited *working memory (WM).* The operation is called *windowing.* Windowing is easier when the blackboard is partitioned in hierarchical domains. The WM data are directed to specialized agencies by enabling commands (*pronome* function of Minsky, 1986). Only one real world and one intended world were represented in my android arm prototype (Callataÿ, 1986), but systems with controlled hypotheses may be designed.

From predicates to keywords: The conditions of an activated rule are a set of simultaneously true predicates. The predicates of a production rule system are computed from variables produced by external inputs and actions of internal rules. In production rule systems, most actions and input data are binding of values to variables in the blackboard. Instead of using the values to compute predicates, one can immediately compute the predicates with enlarged rule actions as shown below.

Like a predicate which is true or false, a keyword is instantiated or not. A cell and its dedicated wire (axon) represents a keyword. A logical impulse in a keyword wire represents its instantiation. In the DNPS model with a cell per rule, a rule produces only one action which may become several actions and keywords when distributed to other agencies by the axon. Then the items in the blackboard are no longer values of variables, but instantiated keywords. The sensory input data are transformed in all-or-none symbols by feature extractors. The actions are commands (impulses) to the external world (servo-mechanism set-points or blackboard window positioning) and thus are keywords. *Predicates disappear in the DNPS architecture.*

Variables: In a context where the agent and the object are animals, the following rules are equivalent with implicit variables:

Indexed document	**Relational rule**
if strong_agent &	*if force(X, strong) &*
carnivore_agent &	*feeding_habit(X, carnivore) &*
weak_object	*force(Y, weak)*
then agent_eats_object	*then eat(X, Y)*

The capabilities of various expert systems depend on the complexity of rules which can be processed and on the possibility of submitting queries to other expert systems from a rule. A complex formalism permits writing an expert system with less rules, because the rules can be made more general.

Some researchers have noticed that complex rules sometimes generate complex inferences with unexpected, false conclusions. The problem may come from generalized rules having unforeseen exceptions. Proponents of simple rule formalism claim to have more robust systems. In these simple rule systems, solving a complex application is more difficult: it is solved with queries called from the simple rules, i.e. it requires a hierarchy of domains, each one with its knowledge base and variables. Rules are better verified within a limited domain, but writing complex expert systems in a simple environment may require too many rules for an efficient inspection. The above issues are not yet clarified.

In the above division of expert systems, the DNPS systems have simple rules, few variables, but may have many hierarchical levels. A DNPS subset, sufficient to solve a query in a limited domain, cannot simultaneously use several variables in the same role, because this is incompatible with parallelism. The rules: *X is taller than Y* and *X is shorter than Z* cannot be processed in a simple DNPS with few domains. A young child cannot find who is the shortest of these three persons (Burt's test). This problem is solved later on, allegedly when additional brain areas have matured, making available metalevels of DNPS hierarchy. The rule with two variables $X = Y$ is also difficult to process in a DNPS. DNPS must be developed with a particular style of programming, as shown in some prototypes (Callataÿ, 1986). Logic does not need explicit variables (Quine, 1966).

Processed data may contain several instances of a property for different variables, like the agent and the recipient, therefore the keywords representing the same property for different variables must have a different virtual name. Handling negative predicates presents unresolved difficulties. Some negative properties can be handled as positive ones (Callataÿ, 1987c).

The Aristotle's categories (roughly corresponding to the categories allowed in a sentence) are a starting point to guess how many variables are needed for human reasoning. Many data are already collected by cognitive scientists to progress in this new field of research. The results should explain why men cannot easily solve some puzzles which are easily processed by computers, and conversely. DNPSs integrate knowledge and processing in each RCAM, contrary to von Neumann machines where memory and CPU are separated. In RCAM all input data are associated and therefore only the missing input can be abstracted. Thus DNPSs cannot directly abstract data, but need a metalevel hierarchy for progressive abstraction. It is conjectured that a sufficient number of metalevels must be present for abstract reasoning in man (Callataÿ, 1986). This may explain why men spontaneously process syllogisms with concrete methods (Johnson Laird, 1983).

Built-in and learned DNPS programs: In the translation of a rule into a DNPS, a source language is compiled into an object code describing a DNPS hardware (the innate part) and some latch positioning (the learned part). In the neural mapping, the left hand side (LHS) of a rule is the cell dendritic tree; the right hand side (RHS) is its axon. More precisely, the LHS is a dendritic tree and its efferents, and the RHS the initial axon. Conflict resolution is done by arbitration of the activated cells of a group.

An RCAM is a simple device exhibiting parallel processing as in *associative networks* (reviewed in Findler, 1979) and in *high interaction systems* of Minsky (1986). A DNPS and a RETE network (Forgy, 1982) have functional similarities (figure 4): with DNPS, the axons distribute the conditions of the working memory and reactivate all existing conditions in parallel. With RETE, the memorized conditions stay active, except those connected by the RETE network to the variables just modified in the blackboard. Only DNPS has a neural mapping.

Roughly, the rule base corresponds to the LTM, the blackboard to the STM, the working memory to the present thought activity. The blackboard must only contain stable states of the external world. Data from the episode memory can be loaded into the short term memory to solve a present problem by analogy. It seems that this event reconstruction was preferred to a large STM by natural selection, possibly because the control of a *blackboard* is too complex to be safely managed by automated systems.

Clauses: A clause is processed twice by a Prolog interpreter: first with calling goals, then with asserted data. In the DNPS model, a clause is replaced by two associated rules: the goals are processed on the top-down network, and the assertions on the bottom-up network. Corresponding goals and assertions are located in a functional column linking the two networks. A column stores a simple behavior rule (clause head or subclause) which is a node in a hierarchy. Structures of variables in a clause are represented by atoms but are predicates:

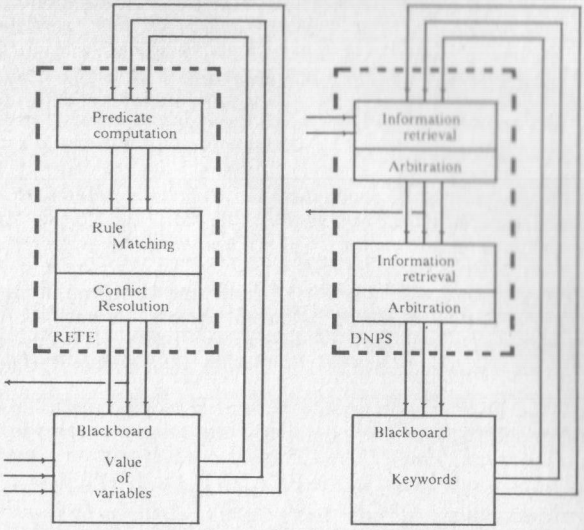


Figure 4. Comparison of the DNPS and RETE network: *In the production rule systems on the left, the blackboard contains values of variables. In the DNPS represented on the right, the blackboard contains only keywords. A RETE network may be represented as a program flowchart. A DNPS is a hardware network corresponding to a conceptual parallel flowchart.*

ch(a,h(X,Y)) if sc1(b,X) & sc2(c,Y).

is replaced by a goal rule:

sc1(b,) & sc2(c,*) if ch(a,*).*

and several asserted rules:

ch(a,z) if sc1(b,x) & sc2(c,y).

where *x*, *y* and *z* are instantiated keywords: *t(X) is renamed x in the present environment.* Each subclause of a clause body is concurrently processed in the DNPS hierarchical branches. This method restricts the use of common variables in AND branches. The innate program built in the functional cortical columns of figure 2, 3 and Callataÿ (1986) is a Prolog-like interpreter with breadth-first search and partial fuzzy matching. I do not know whether a hierarchy of DNPS pattern matching may correspond to some cases of unification methods. The focussing mechanism automatically restricts the search space per period to one instance per category (a property of RCAM classifiers).

Three analogous models: The three following models have a functional similarity: I conjecture that most computation done with a model may also be done on others, but their possible extensions are different.

- 1) a production rule system, using assignments of variables.
- 2) a hierarchical system of information retrieval, using keyword activations.
- 3) a DNPS of content addressable memory, using impulses.

The first model has a large scope of applicability, because it can be mixed with traditional programming. It is used for expert systems where it has been studied. The second model makes partial matching, finding analogous rules when the first model is halted (Callataÿ, 1987c). The brain does not seem having circuits noticing which computation method was used in RCAM: complete or partial matching (I call this last

method *intuition*). The third model is directly executable in logic circuits. It handles fuzzy data and connectionist learning without additional devices visible on the machine diagram. It is easily extended for parallelism because the production of knowledge instances is partitioned and categorized by the DNPS architecture.

Safety of parallel operations: A DNPS defines a simple working memory: one element per node, or one element per loop. A matching is computationally safe if it matches one level of working memory - one coherent situation - against many rules in parallel. Knowledge partitioning prevents the production of two entities in the same category. The architecture prevents the creation of two such entities at a time by arbitrating possible competition. Focussing on several contexts in stacked working memories is not safe, as two keywords in the same category could be processed. The concurrent matching of many patterns to many objects is not safe. The RETE matching algorithm in Hillyer and Shaw (1986) is partially parallel (it is executed in several cycles). A parallel production rule system with fuzzy data was developed by Siler et al. (1987). It seems possible to use redundant data, but with some constraints and computational risks (Callataÿ, 1986).

Compilation: The DNPS model should permit developing a special compiler for a subset of OPS or Prolog programs, like compilers translating programs to Programmable Logic Automata (Hartmann, 1986). The compiled product will be a machine structure (DNPS) and a set of relations stored in the RCAMs of the DNPS nodes. This resulting code should simulate or build a machine running the program. A program compilation produces a hardware requirement which cannot be reduced to run on a smaller DNPS machine. The structure of the source program will still be apparent in the resulting machine structure. Some programs should have a simple mapping - those easily solvable by humans: others may be almost impossible to implement - those with a difficult reasoning requiring a large short term memory.

Overall program: The deductive operations done during a period give an accurate solution to a *"problem"* focussing on a database partition and an STM window. Only some types of problems (those definable with generic types) can be solved in one period or epoch (an epoch is the time for which a top episode cell is held).

The chaining of periodic DNPS operations is not yet studied. The following suggests a few possible directions. The behavior of an autonomous intelligent robot consists of a sequence of problems generated by the previous event and taking account of the changing body environment. These operations can be managed by a DNPS processor using a knowledge metalevel. This last system may work as an almost independent expert system, act with concepts taken at a high level of abstraction, activate blackboard windows and set low level intentions. With mechanisms focussing on interesting problems and being rewarded for successful actions, an "increasingly rewarded behavior" should develop in a well-designed and well-trained robot.

Behavior and logic: According to working hypotheses, animals record their behavior (episodes). These episodes taken as a whole are the only rules which are logically true in a repetitive world (some additional assumptions are needed for a theory: sensory input of the relevant data, categorical perception without significant information loss).

By contrast, the scientific approach consists of applying general rules explaining the world. Most rules are numerical formulas - physical laws - which are not directly processable by symbolic computation. The discovery of generalized rules is made by induction, but, because rules induced from a few examples may become false later on, the system is hardly compatible with an additive memory.

As a conclusion of these two approaches, science has few simple rules (axioms), while behavior has many complex rules which are logically inapplicable because identical situations normally never happen again.

Here, the robot method is not to discover generalized rules. For each new situation, the robot recognizes the most similar episodes. With partial matching, recognition can still be successful when significant data are missing and additional data (noise) are present. Thus a goal is logically solved when the incomplete and/or noisy data do not influence the action issue, i.e. when the analogy or metaphor is valid. Because knowledge accumulation does not prevent selection of the most similar event with the RCAM regular processing speed, behavior becomes more frequently successful when more relevant rules have been learned.

Men choose their top intentions with simple rules such as "habits", but these acts are arbitrary (the so-called freedom of will) and should not be confused with finding how to execute a given intention, the only problem where logic may be involved. It is a safe feature that the taught generalized rules cannot be transferred in the main knowledge base for spontaneous processing (these rules may be progressively transferred, but only when they are experienced in real life). The intentional method allows most vertebrates to check before action that their intentions will possibly generate a rewarding behavior, and not a threatening one.

Conclusions

The performances of symbolic neural networks (DNPS) have been tested with an android arm prototype (Callataÿ, 1986). The robot has spontaneously learned how to reach a target with very fast movements. The training was quite short. Other prototypes have shown the efficiency of some classification algorithms. Hierarchical networks have not yet been tested, but the above description shows that well-formed DNPSs provide logical deductions when situations can be perfectly matched. Partial matching seems to be a computational method for reasoning by analogy, a method whose relevance is asserted by human behavior.

Adaptation to a changing physical world is easily done by changing the context. The method explains why persons, who have learned to ride a bicycle with goggles reversing vision, can at once continue riding without goggles.

In summary, symbolic neural networks are "gedanken" architectures having at present few explanatory theories (although DNPSs have more logical justifications than adaptive neural networks). I do not know a theory explaining why reasoning by analogy is frequently efficient and thus whether partial matching is worthwhile. Experimentation is the main open research route, but it is also worth investigating whether new (or existing theories) can show the DNPS limitations.

The large DNPS machines remain conceptual because VLSI technology cannot yet build RCAMs at affordable prices. According to working hypotheses, an RCAM cycle (an information retrieval query with fuzzy data on a rule base of one million weighted indexes) can be simulated in 15ms in a 1 Mips microprocessor. A biological brain is then as fast as this system (Callataÿ, 1986). A human brain may contain about one million RCAMs equivalent to the above microprocessors and accumulate 10^{11} simple rules at the rate of a few million rules per day. A convincing prototype would be an artificial animal exhibiting spontaneous learning. Developing a sensory/motor interface for this robot may be more complex than building a large DNPS.

Acknowledgement: *This work was done with the cooperation of M. Sintzoff, C. Trullemans, and C. Veraart. I thanks them and also D. Ribbens, and L. Steels, for their advices. In a conversation, P. Smets helped to develop several points. I am also grateful for the comments of M. King and D. Crowe.*

The views and opinions herein are solely those of the author, who also takes responsibility for any error or misconception.